User's Guide

CC2564x Demo Applications

Jesus Pintado

Table of Contents

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

The TI dual-mode CC2564 Bluetooth stack for MSP432 and STM32 MCUs includes sample applications which demonstrate the use of TI's Bluetooth stack. This guide will show how users can use and interact with each specific application available. For general information about the stack for the respective MCUs refer to the CC2564C TI Dual-Mode Bluetooth Stack on MSP432 MCUs and the CC2564C TI Dual-Mode Bluetooth Stack on STM32F4 MCUs.

2 A2DP (AUD) Demo Guide

2.1 Demo Overview

Note

These instructions can be used to run this demo on the either the STM32 (AUD Demo) or Tiva (A2DP Demo) Platforms, but the STM32F4 only supports the SINK role.

The Advanced Audio Distribution Profile (A2DP) Sink and Audio Video Remote Control Profile (AVRCP) allows a device to act as an Audio sink and can control and stream audio on an Audio source. It is recommended that the user visits the kit setup Getting Started Guide for STM32 or Getting Started Guide for TIVA pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, look at the Device manager for Communications Port (COM x) under Ports (COM & LPT). Attach a terminal program like PuTTY to the serial port (COM x) for the board, x means which COM is open for Communications Port in Device Manager. The serial parameter to use is 115200 Baud rate. Once connected, reset the device using Reset button (The black button near the blue buttons) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands. This device will become the server.

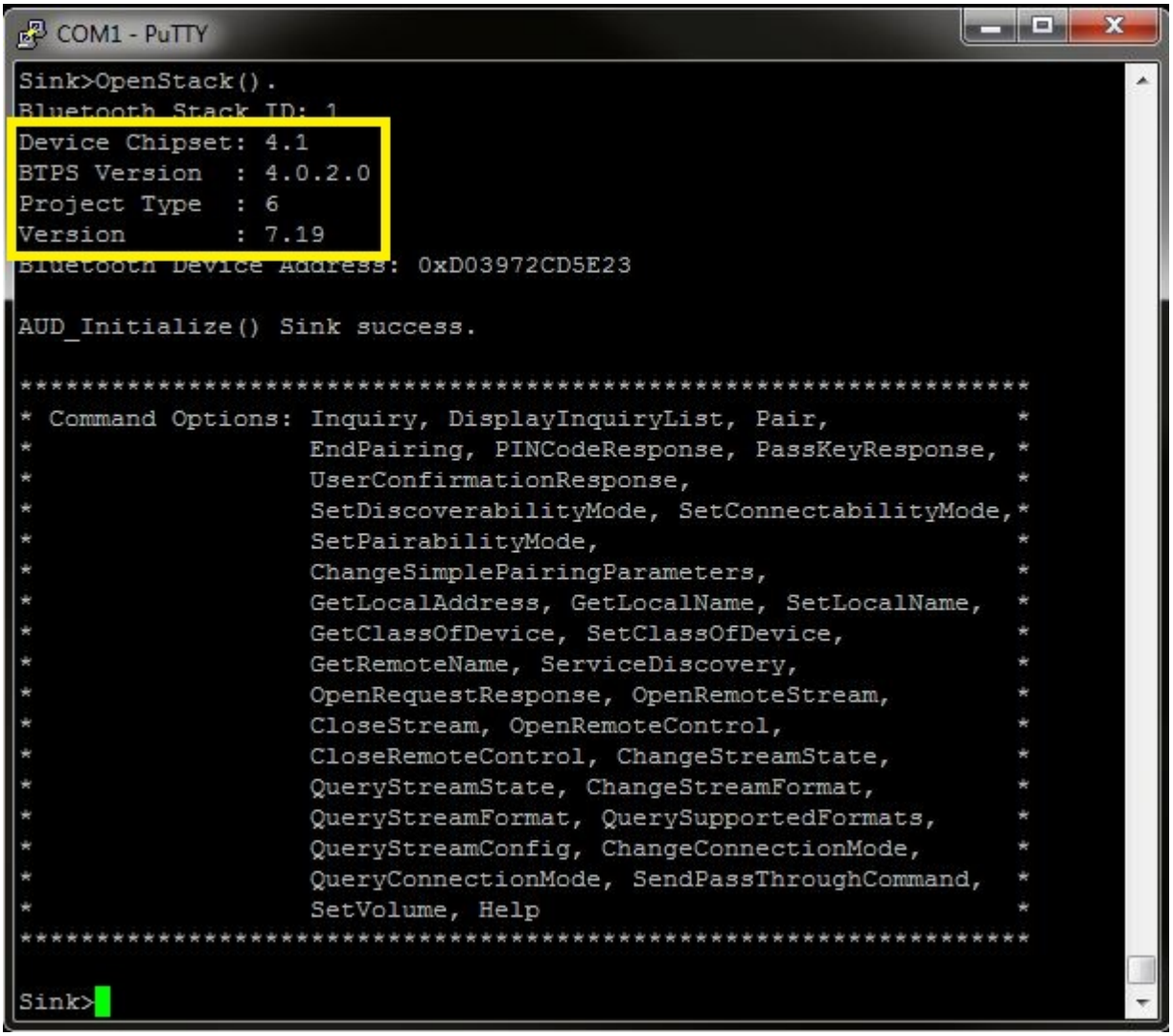

Figure 2-1. AUDDemo Start Screen

Note

The yellow square holds the FW and BTPS versions for future use.

2.2 Demo Application

This section provides a description of how to use the demo application to connect an audio source to it and communicate over Bluetooth.

Device (sink) setup on the demo application

The STM32 board is ready to connect right after it starts, to create this connection, either the source or the sink can initialize it.

Initiating connection from the source

The A2DP source can be any application that can transmit audio. For our example, we will use an android phone as the demo. In order to connect to the sink, the source needs the sink Bluetooth address or name.

Sink Terminal

1. The Bluetooth address can be found at the beginning of the demo application or by typing GetLocalAddress, in this case we can see the bluetooth address.
2. The Sink name can be found by typing GetLocalName and the terminal will print the default name of the application AUDDemo

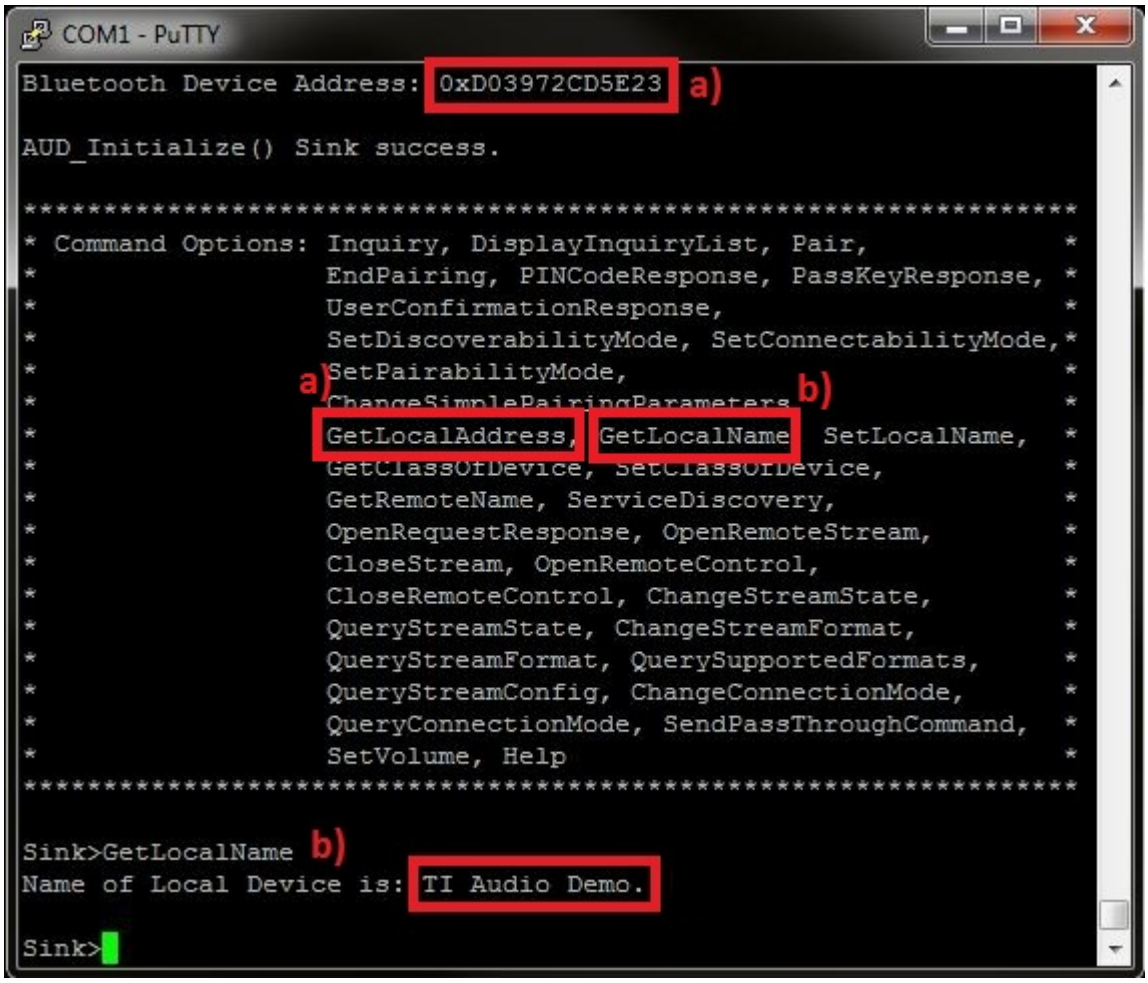

Figure 2-2. AUDDemo Name and Address

Source Phone

After the data of the sink for the connection is known:

3. Open the Bluetooth settings menu on the android phone (Settings->Bluetooth).
4. Hit on search for devices. The phone should begin looking for other Bluetooth devices.

Figure 2-3. AUDDemo Searching for Devices

5. An AUDDemo or A2DPDemo should appear like shown below in the picture. Click on the device to begin pairing.

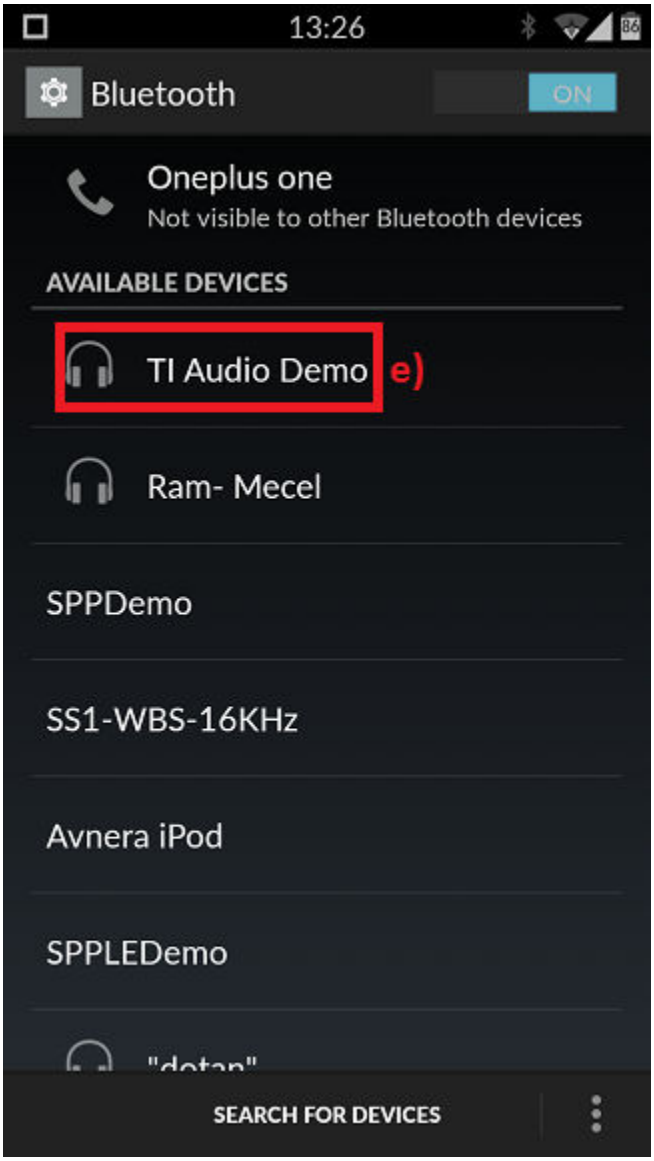

Figure 2-4. AUDDemo TI Device Name

6. Enter PIN password and remember it, in the example we use 0000.

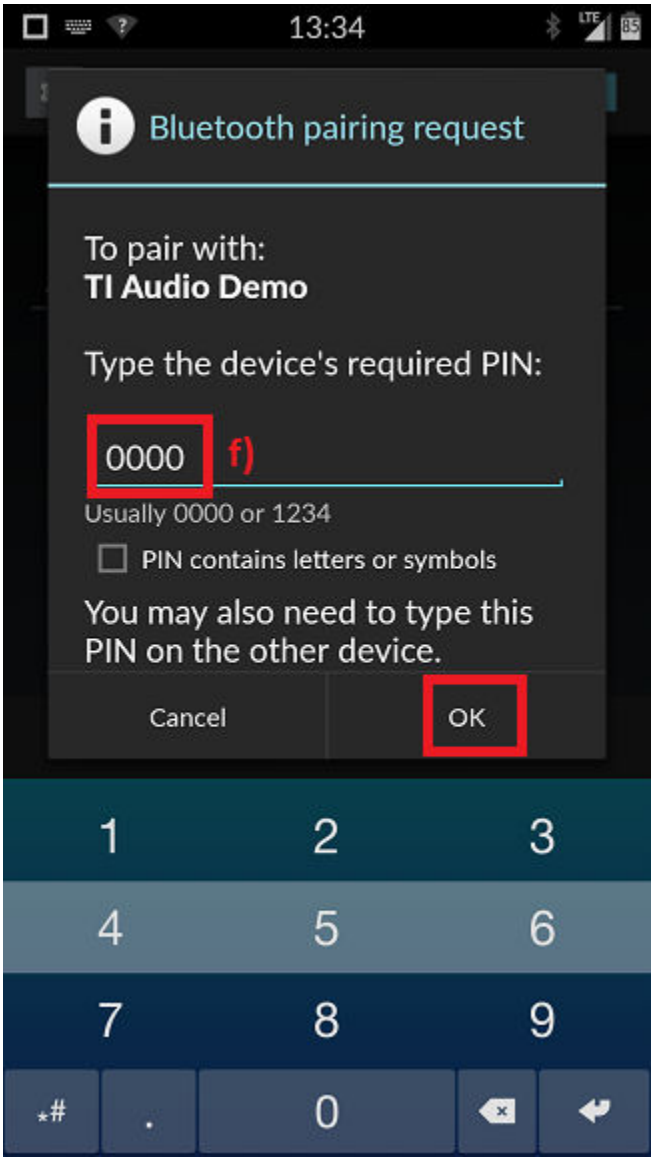

Figure 2-5. AUDDemo Demo Pin

Sink terminal

7. In the Sink terminal there is a request for PIN code that to allow the pairing to complete, enter PINCodeResponse 0000.

Figure 2-6. AUDDemo Pin Code Response

8. After the devices are paired, the device should show connected on the phone side and on the STM32.

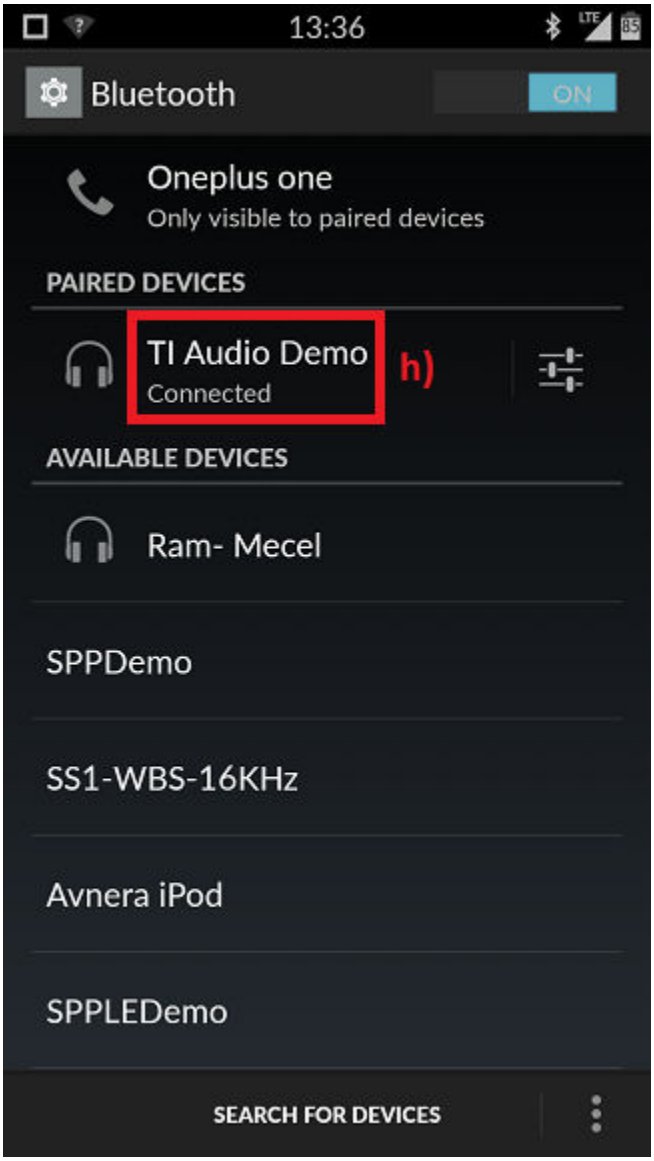

Figure 2-7. AUDDemo Phone Connected

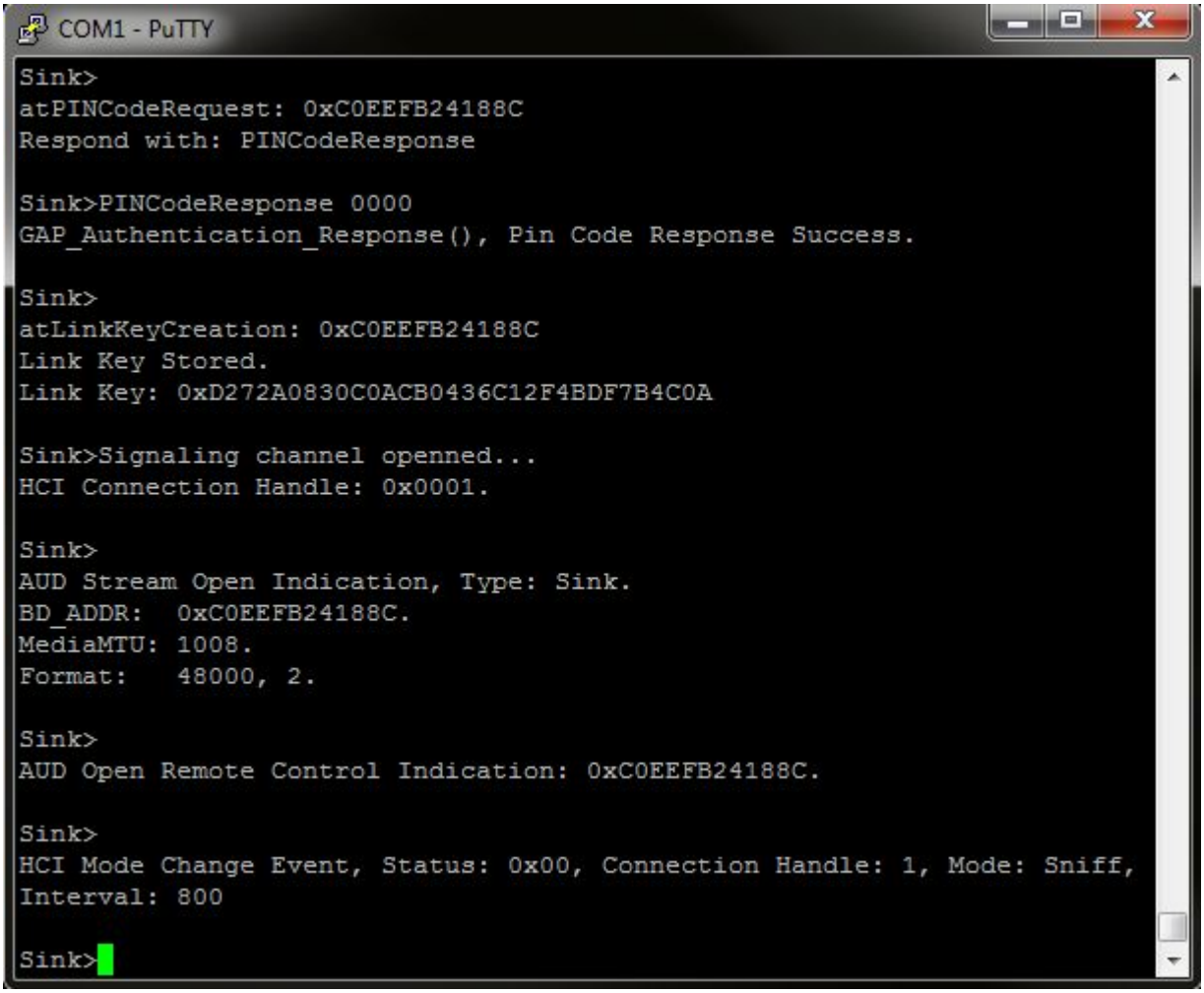

Figure 2-8. AUDDemo Connected From Source

9. You can now control the audio of the phone from the STM32 board.

Initiating Connection from the Sink

Note

Make sure that the source is in discoverable mode

1. In Order to connect to the source, the sink must find it first with inquiry command, type inquiry.

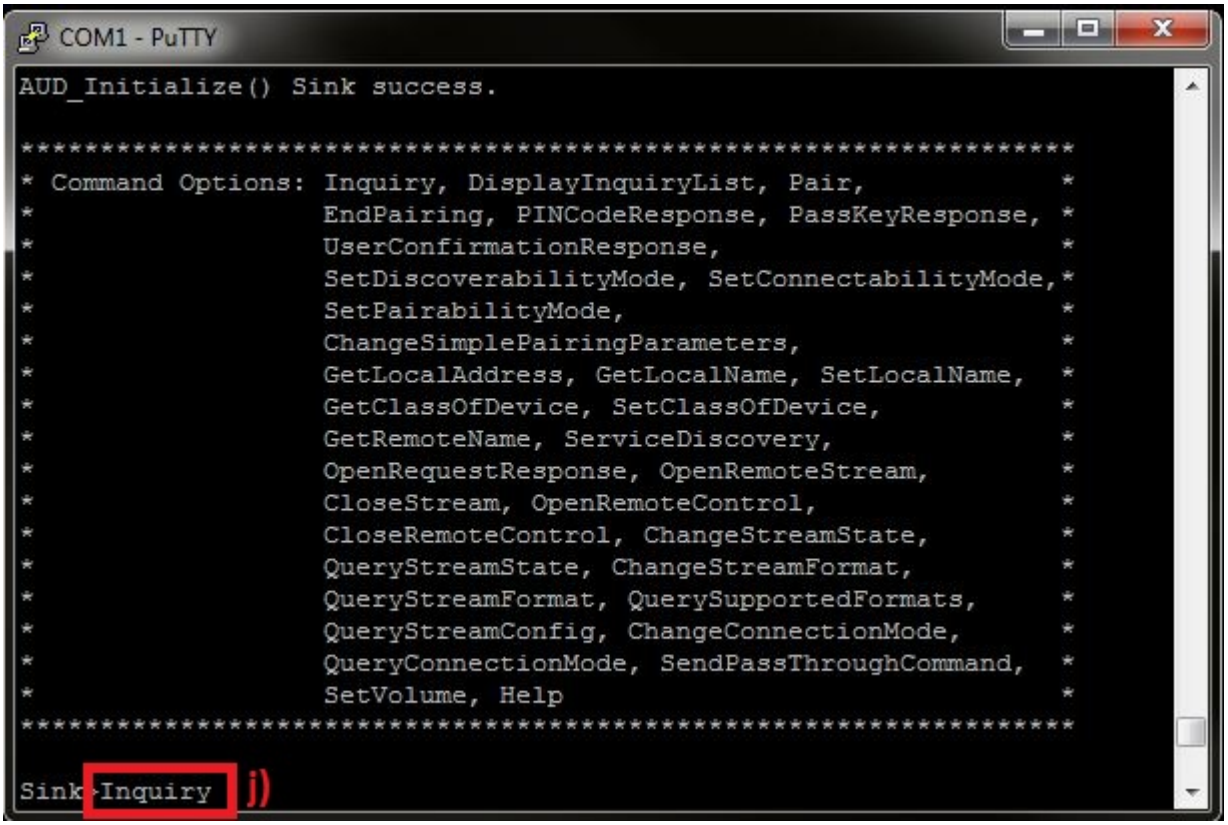

Figure 2-9. AUDDemo Inquiry Sink

2. The inquiry command generates a list of all the devices in range.
3. In order to connect to the source use the OpenRemoteSteam command with the index of the bluetooth address you want, in our case we connect to index 2, so we type **OpenRemoteStream 2**.

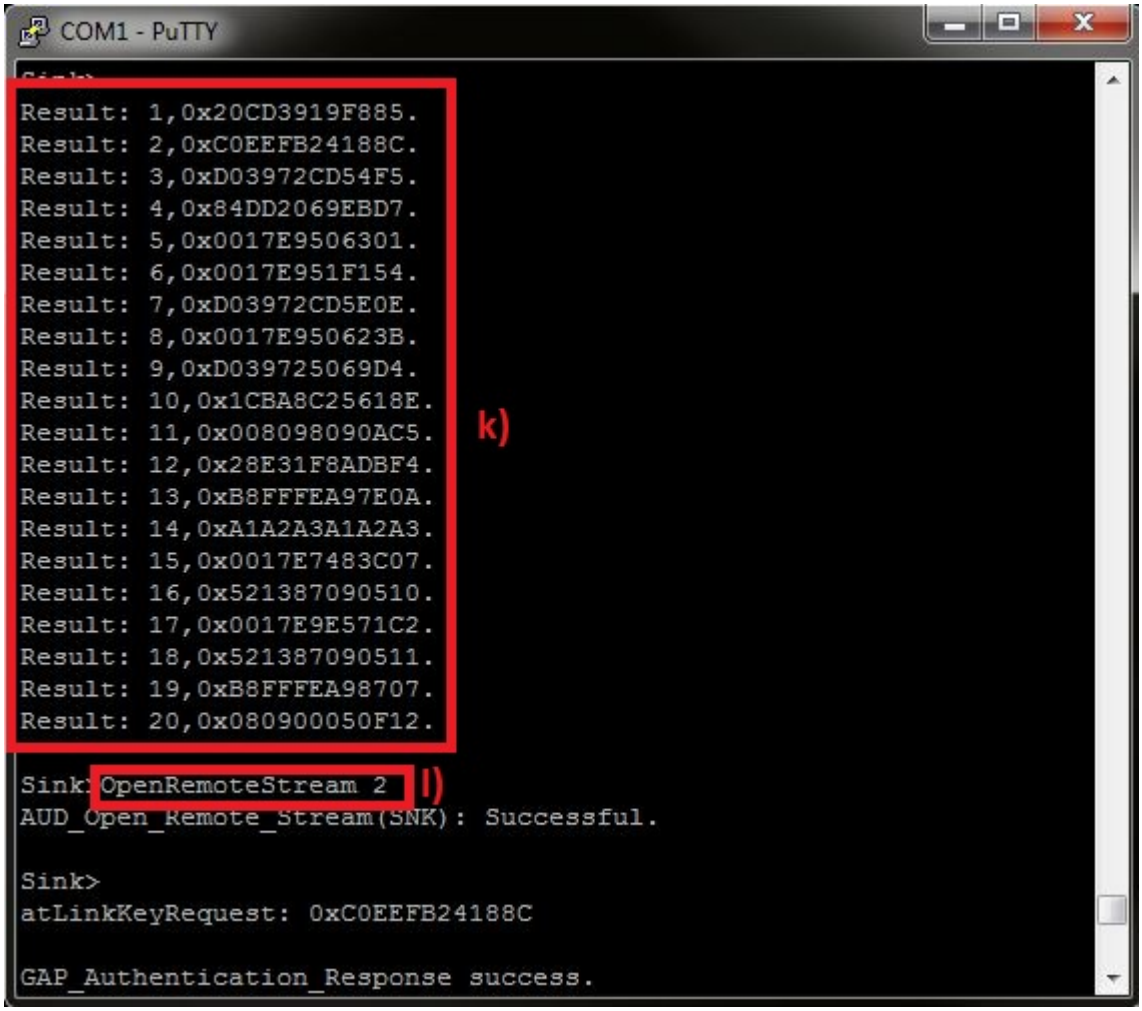

Figure 2-10. AUDDemo Open Remote Stream

Sink Terminal

4. Enter PIN password and remember it. In the example we use 0000.

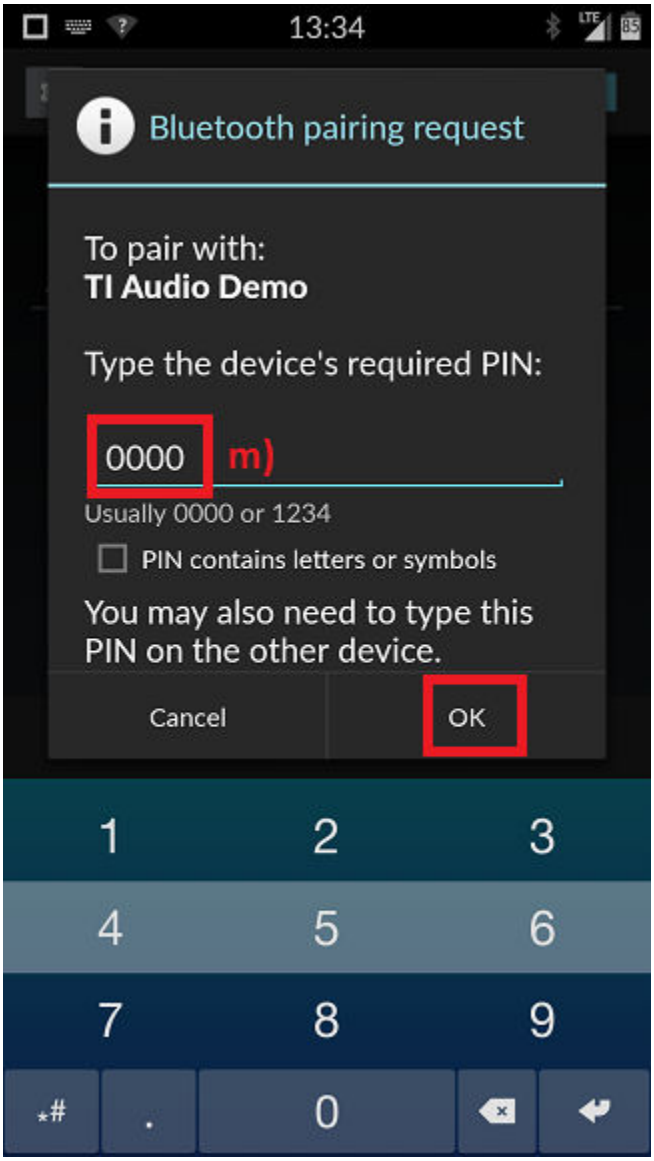

Figure 2-11. AUDDemo Pin Sink

Sink terminal

5. In the Sink terminal there is a request for PIN code that to allow the pairing to complete, enter **PINCodeResponse 0000**.

Figure 2-12. AUDDemo Pin Code Response Sink

6. After the devices are paired, the device should show connected on the phone side and on the STM32.

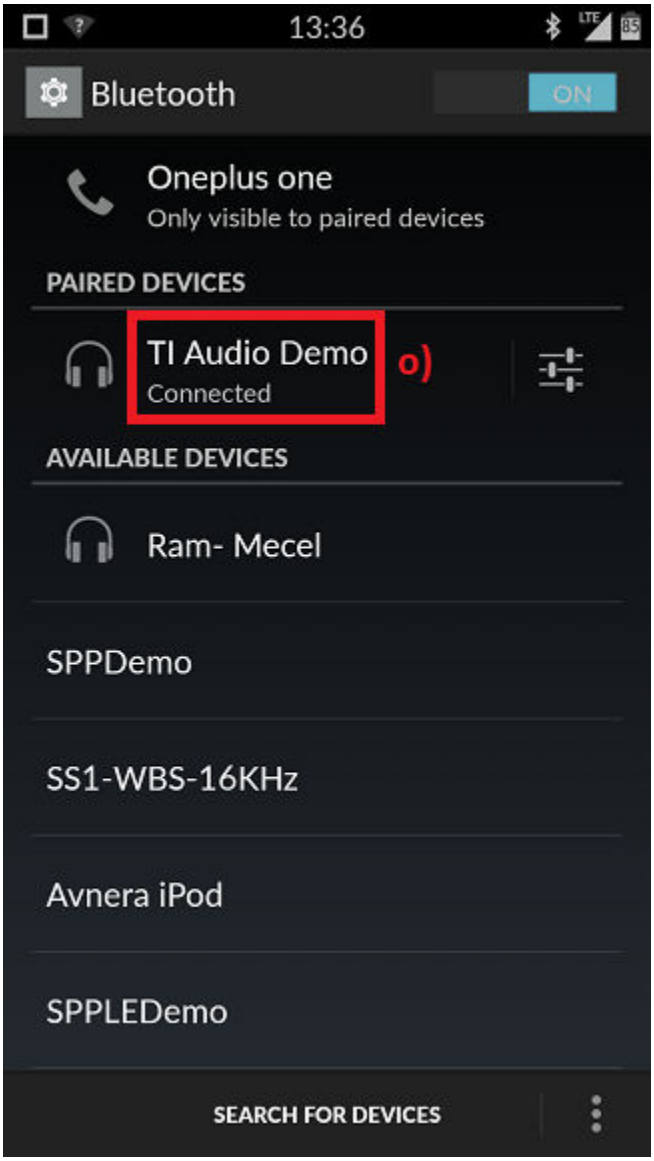

Figure 2-13. AUDDemo Phone Connect from Sink

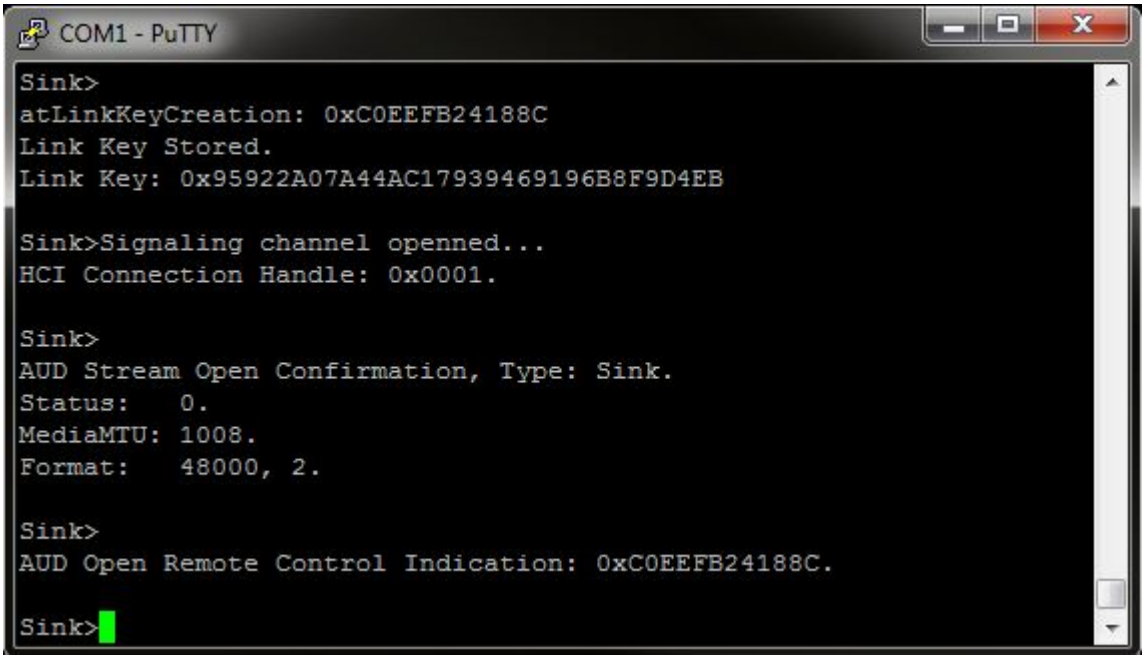

Figure 2-14. AUDDemo Connected From Sink

A2DP+AVRCP Commands

Now, after the sink and the source are connected we can start controlling the audio of the source from the sink.

1. Open any music application in the source, in the sink, in order to control the audio use **SendPassThroughCommand** Command and a parameter for the command. the command can be **Pause = 0, Play = 1, Stop = 2, Vol. up = 3 and Vol. Down = 4**.

> **Note**
> Volume (Vol.) commands might not work with your device.

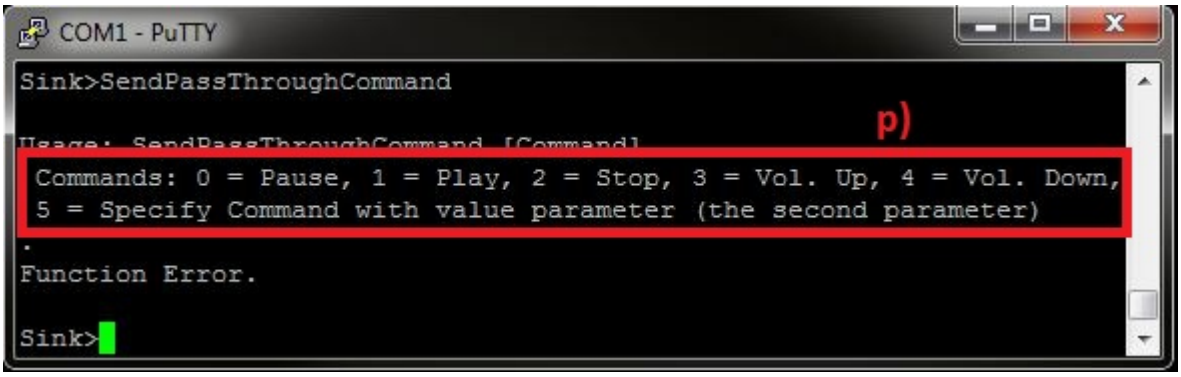

Figure 2-15. AUDDemo Send Pass Through Command

2. Type **SendPassThroughCommand 1**. A track will start playing on the devices.

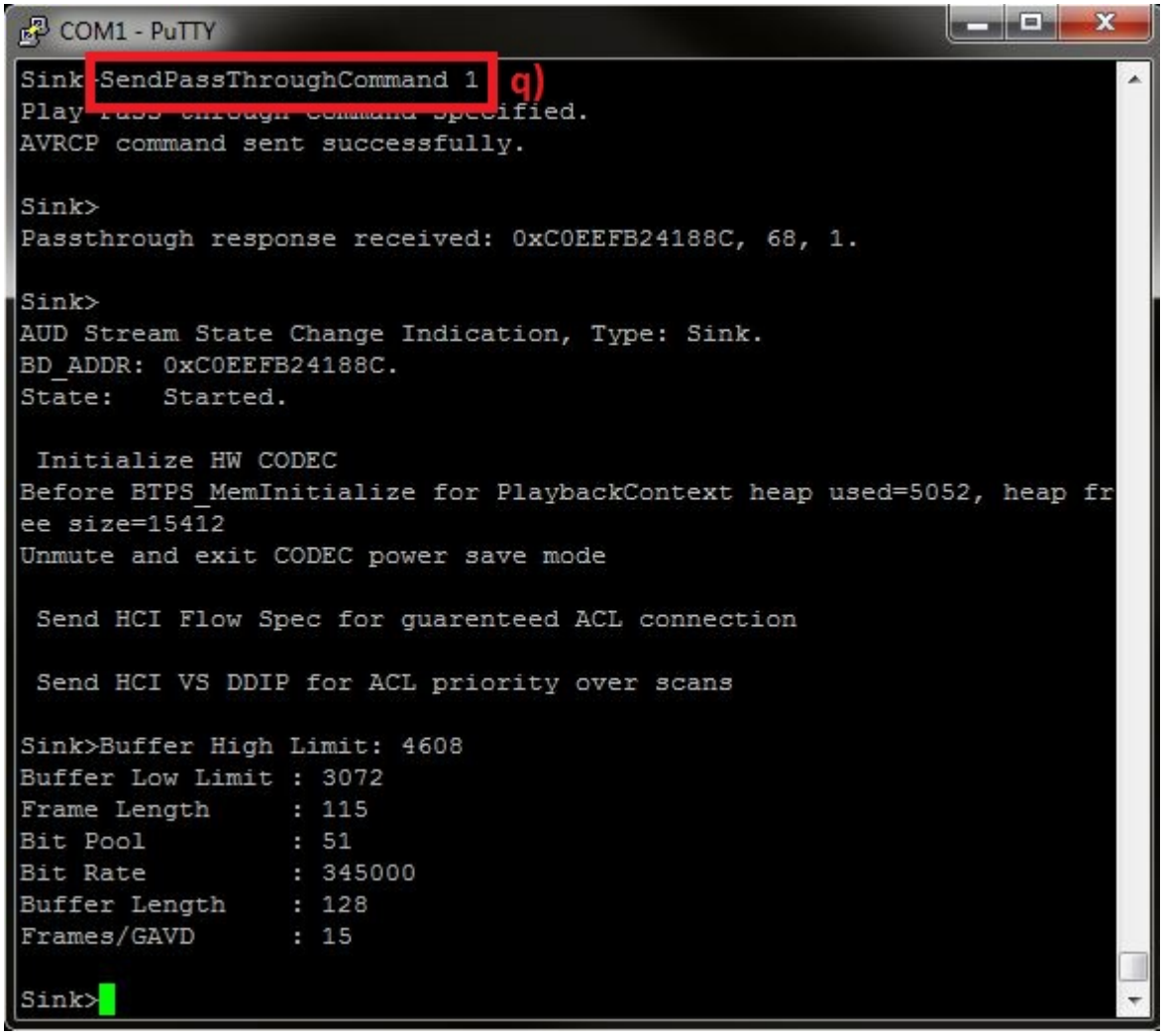

Figure 2-16. AUDDemo Send Pass Through Command 1

3. Type SendPassThroughCommand 2. A track will stop playing on the devices.

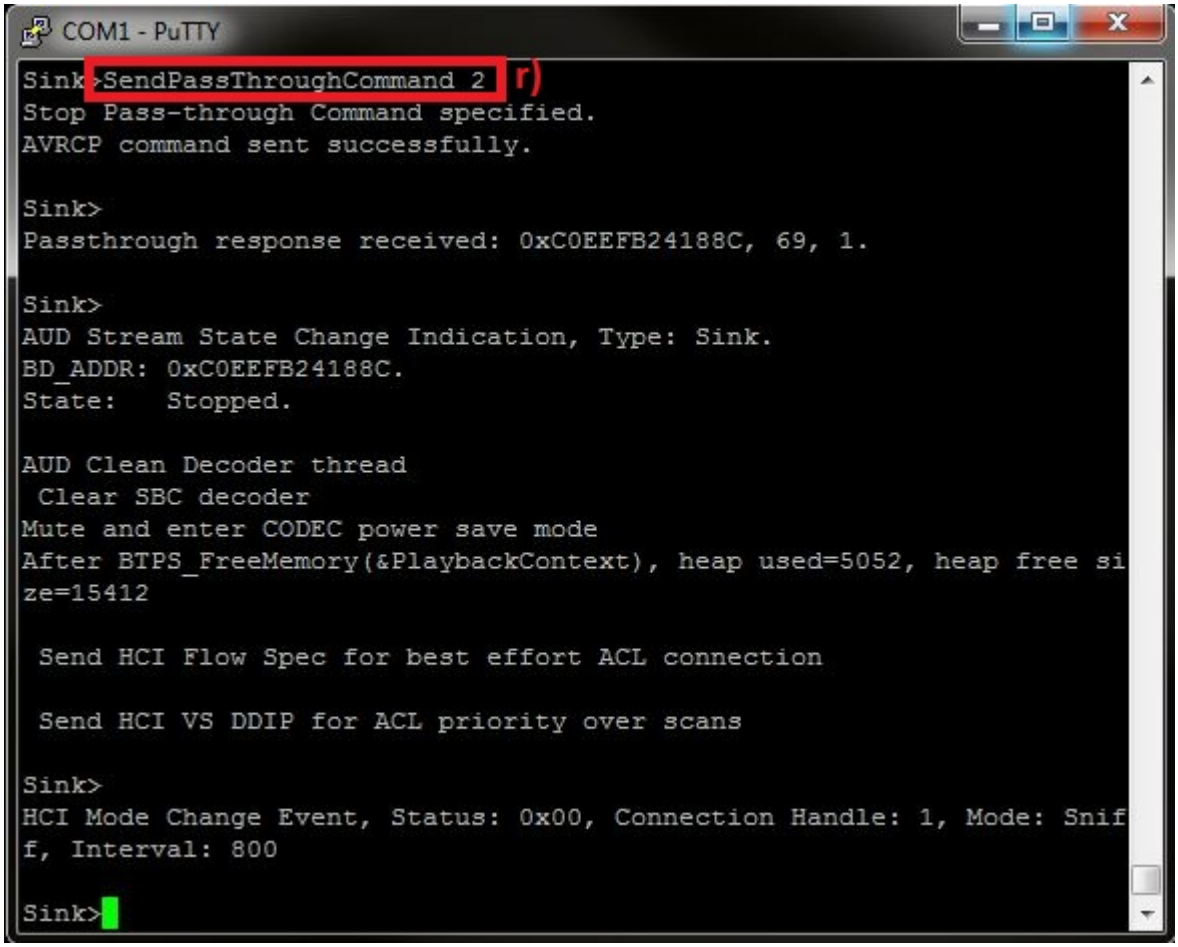

Figure 2-17. AUDDemo Send Pass Through Command 2

4. In order to change the Stream format: Sample rate and Mono/Stereo, use the **ChangeStreamFormat** command and a value which represent the supported format index that is needed.
5. The supported sink formats and their indexes can be displayed by typing **QuerySupportedFormats**.

Note

The connection must be in suspend in order to work. ChangeStreamFormat might not work with your device.

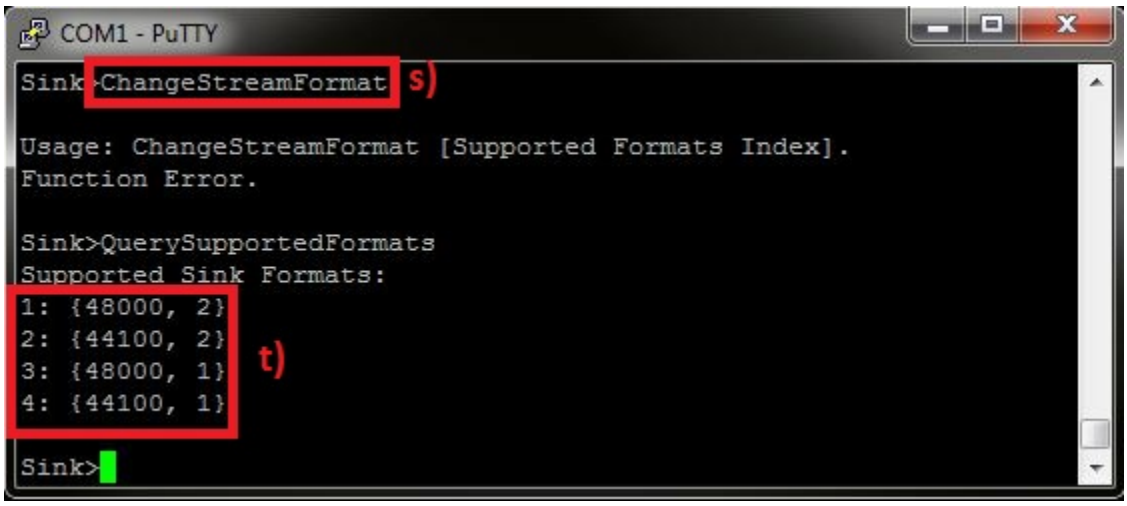

Figure 2-18. AUDDemo Change Stream Format

The First Number is the index that you choose, the second number is the sample rate in Hz and the third number represents **1 = Mono, 2 = Stereo**.

6. In order to change the Stream State: Stopped or Started, use the **Changestreamstate** command and a value which represent the stream state **0 = Stopped, 1 = Started**. When the Command Changestreamstate 0 is sent, the devices will enter to suspend mode of A2DP.

Note

Changestreamstate might not work with your device.

Figure 2-19. AUDDemo Change Stream State

7. The command ChangeConnectionMode enable the sink to allow automatically or manual connections or disallow connection, use the ChangeConnectionMode command and a value which represent the connection mode **Automatic Accept = 0, Automatic Reject = 1, Manual Accept = 2**.

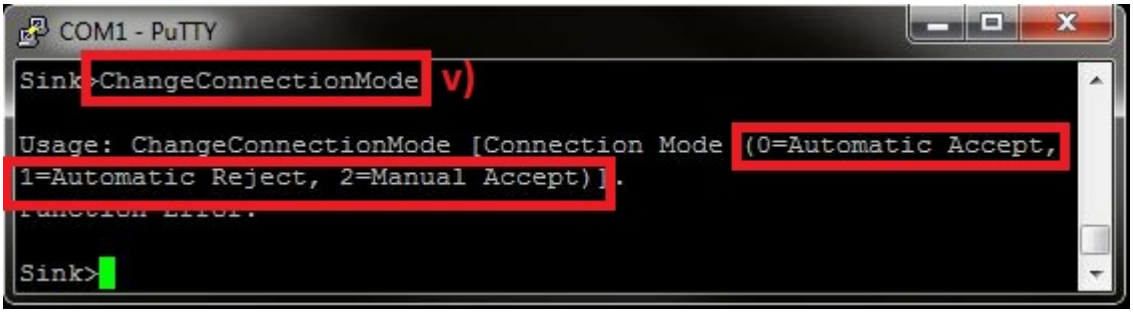

Figure 2-20. AUD Demo Change Connection Mode

2.3 Application Commands

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands

A2DP Profile Comands

OpenRemoteSystem

Description

The following function is responsible for initializing AUD if necessary, and initializing the A2DP and AVRCP subsystems.

Parameters

This command takes Inquiry Index number to work which can be found using the DisplayInquiryList command after an Inquiry has been completed

Possible Return Values

(0) AUD Sink opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2000) BTAUD_ERROR_INVALID_PARAMETER

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2009) BTAUD_ERROR_STREAM_NOT_INITIALIZED

(-2010) BTAUD_ERROR_UNABLE_TO_CONNECT_REMOTE_STREAM

(-2011) BTAUD_ERROR_STREAM_ALREADY_CONNECTED

(-2013) BTAUD_ERROR_STREAM_CONNECTION_IN_PROGRESS

API Call

AUD_Open_Remote_Stream(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam - 1)], astSNK)

API Prototype

int BTPSAPI AUD_Open_Remote_Stream(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, AUD_Stream_Type_t StreamType)

Description of API

The following function is responsible for opening a remote streaming endpoint on the specified remote device.This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the requested Audio Manager is present, followed by the remote Bluetooth device AND the local Stream type.

This function returns zero if successful or a negative return error code if there was an error.

OpenRemoteControl

Description

The following function is responsible for opening a remote control connection to a remote Bluetooth Device, AVRCP subsystem.

Parameters

This command takes Inquiry Index number to work which can be found using the DisplayInquiryList command after an Inquiry has been completed

Possible Return Values

(0) AVRCP opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2000) BTAUD_ERROR_INVALID_PARAMETER

(-2026) BTAUD_ERROR_REMOTE_CONTROL_NOT_CONNECTED

(-2028) BTAUD_ERROR_REMOTE_CONTROL_ALREADY_CONNECTED

(-2029) BTAUD_ERROR_REMOTE_CONTROL_CONNECTION_IN_PROGRESS

(-2030) BTAUD_ERROR_REMOTE_CONTROL_NOT_INITIALIZED

API Call

AUD_Open_Remote_Control(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam - 1)])

API Prototype

int BTPSAPI AUD_Open_Remote_Control(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR)

Description of API

The following function is responsible for opening a remote control connection to the specified remote device. This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the requested Audio Manager is present, followed by the remote Bluetooth device address of the device to connect with. This function returns zero if successful or a negative return error code if there was an error.

Note

If this function is called to establish a remote control connection (and the connection is successful), the caller ***MUST*** call the AUD_Close_Remote_Control() function to disconnect the remote control connection

CloseRemoteControl

Description

The following function is responsible for closing an opened remote control.

Parameters

This command takes Inquiry Index number to work which can be found using the DisplayInquiryList command after an Inquiry has been completed.

Possible Return Values

(0) AVRCP closed successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-2000) BTAUD_ERROR_INVALID_PARAMETER

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2005) BTAUD_ERROR_INVALID_OPERATION

(-2026) BTAUD_ERROR_REMOTE_CONTROL_NOT_CONNECTED

(-2030) BTAUD_ERROR_REMOTE_CONTROL_NOT_INITIALIZED

API Call

AUD_Close_Remote_Control(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam - 1)])

API Prototype

int BTPSAPI AUD_Close_Remote_Control(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR)

Description of API

The following function is responsible for closing a currently open remote control connection on the local device.This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the requested Audio Manager is present, followed by the device address of the remote control connection close.This function returns zero if successful or a negative return error code if there was an error.

Note

This function should only be called if the local device previously issued a call to the AUD_Open_Remote_Control() function and a remote control connection was successfully established.

ChangeStreamState

Description

The following function is responsible for changing the state of an open stream.

Parameters

This function required one parameter, stream state (0 = Stopped, 1 = Started).

Possible Return Values

(0) Successful

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2000) BTAUD_ERROR_INVALID_PARAMETER

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2012) BTAUD_ERROR_STREAM_NOT_CONNECTED

(-2016) BTAUD_ERROR_STREAM_STATE_ALREADY_CURRENT

(-2017) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_STATE

(-2018) BTAUD_ERROR_STREAM_STATE_CHANGE_IN_PROGRESS

(-2019) BTAUD_ERROR_STREAM_FORMAT_CHANGE_IN_PROGRESS

API Call

AUD_Change_Stream_State(BluetoothStackID, ConnectedBD_ADDR, astSNK, (TempParam->Params[0].intParam)?astStreamStarted:astStreamStopped)

API Prototype

int BTPSAPI AUD_Change_Stream_State(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, AUD_Stream_Type_t StreamType, AUD_Stream_State_t StreamState)

Description of API

The following function is responsible for Changing the Stream State of a currently opened stream endpoint on the local device. This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the requested Audio Manager is present, followed by the remote device address of the connected stream, followed by the stream endpoint type to change the state of (local Stream Endpoint), followed by the new Stream Endpoint state.

This function returns zero if successful or a negative return error code if there was an error.

ChageStreamFormat

Description

The following function is responsible for changing the format of a currently open stream. The Supported Formats Index should be determined by the displayed list from a call to QuerySupportedFormats.

Parameters

This function required one parameter, Supported Formats Index (fron the list in QuerySupportedFormats).

Possible Return Values

(0) Successful

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2000) BTAUD_ERROR_INVALID_PARAMETER

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2012) BTAUD_ERROR_STREAM_NOT_CONNECTED

(-2014) BTAUD_ERROR_STREAM_IS_ACTIVE

(-2018) BTAUD_ERROR_STREAM_STATE_CHANGE_IN_PROGRESS

(-2019) BTAUD_ERROR_STREAM_FORMAT_CHANGE_IN_PROGRESS

(-2020) BTAUD_ERROR_UNSUPPORTED_FORMAT

(-2021) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_FORMAT

(-2032) BTAUD_ERROR_SAME_FORMAT

API Call

AUD_Change_Stream_Format(BluetoothStackID, ConnectedBD_ADDR, astSNK, &AudioSNKSupportedFormats[(TempParam->Params[0].intParam - 1)])

API Prototype

int BTPSAPI AUD_Change_Stream_Format(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, AUD_Stream_Type_t StreamType, AUD_Stream_Format_t *StreamFormat)

Description of API

The following function is responsible for Changing the Stream Format of a currently opened stream endpoint on the local device. This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the requested Audio Manager is present, followed by the remote device address of the connected stream, followed by the stream endpoint type to change the format of (local Stream Endpoint), followed by the new Stream Endpoint format.

This function returns zero if successful or a negative return error code if there was an error.

Note

The Stream Format can ONLY be changed when the stream state is stopped.The stream codec type cannot be changed. If a stream has been configured for a specific codec type, this function cannot be used to change the codec type.The stream must be disconnected and reconnected and the alternate codec type must be selected.In fact, the codec type bit mask value is ignored by this function.

QueryStreamFormat

Description

The following function is responsible for determining and displaying the format of a currently opened stream.

Parameters

This function required one parameter, Local Stream Type (0 = SNK, 1 = SRC).

Possible Return Values

(0) Successful

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2000) BTAUD_ERROR_INVALID_PARAMETER

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2012) BTAUD_ERROR_STREAM_NOT_CONNECTED

API Call

AUD_Query_Stream_Format(BluetoothStackID, ConnectedBD_ADDR, astSNK, &StreamFormat)

API Prototype

int BTPSAPI AUD_Query_Stream_Format(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, AUD_Stream_Type_t StreamType, AUD_Stream_Format_t *StreamFormat)

Description of API

The following function is responsible for querying the current Stream Format of a currently opened stream endpoint on the local device.This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the requested Audio Manager is present, followed by the remote device address of the connected stream, followed by the stream endpoint type to query the format of (local Stream Endpoint), followed by a pointer to a buffer that is receive the current Stream Endpoint format.

This function returns zero if successful or a negative return error code if there was an error.

QuerySupportedFormats

Description

The following function is responsible for displaying the local supported formats to the user. The indices displayed should be used with the ChangeStreamFormat function.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the query.

Possible Return Values

(0) Successful

(-8) INVALID_STACK_ID_ERROR

QueryStreamConfiguration

Description

The following function is responsible for determining and displaying the configuration of a currently opened stream.

Parameters

This function required one parameter, Local Stream Type (0 = SNK, 1 = SRC).

Possible Return Values

(0) Successful

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2000) BTAUD_ERROR_INVALID_PARAMETER

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2012) BTAUD_ERROR_STREAM_NOT_CONNECTED

API Call

AUD_Query_Stream_Configuration(BluetoothStackID, ConnectedBD_ADDR, astSNK, &StreamConfiguration)

API Prototype

int BTPSAPI AUD_Query_Stream_Configuration(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, AUD_Stream_Type_t StreamType, AUD_Stream_Configuration_t *StreamConfiguration)

Description of API

The following function is responsible for querying the current Stream Configuration of a currently opened stream endpoint on the local device.This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the requested Audio Manager is present, followed by the remote device address of the connected stream, followed by the stream endpoint type to query the configuration of (local Stream Endpoint), followed by a pointer to a buffer that is receive the current Stream Endpoint configuration.

This function returns zero if successful or a negative return error code if there was an error.

Note

This function is used to query the low level A2DP configuration that is currently active for the specified stream.

ChangeConnectionMode

Description

The following function is responsible for changing how incoming connections are handled.

Parameters

This function required one parameter, Connection Mode (0 = Automatic Accept, 1 = Automatic Reject, 2 = Manual Accept).

Possible Return Values

(0) Successful

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2000) BTAUD_ERROR_INVALID_PARAMETER

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

API Call

AUD_Set_Server_Connection_Mode(BluetoothStackID, (!(TempParam->Params[0].intParam))? ausAutomaticAccept:(TempParam->Params[0].intParam == 1)?ausAutomaticReject:ausManualAccept)

API Prototype

int BTPSAPI AUD_Set_Server_Connection_Mode(unsigned int BluetoothStackID, AUD_Server_Connection_Mode_t ServerConnectionMode)

Description of API

The following function is responsible for setting the Audio Manager Server Connection Mode.This function accepts as its first parameter the Bluetooth stack ID of the Bluetooth stack in which the server exists.The second parameter to this function is the new Server Connection Mode to set the Server to use. Connection requests will not be dispatched unless the Server Mode (second parameter) is set to ausManualAccept. In this case the Callback that was registered with the server will be invoked whenever a Remote Bluetooth device attempts to connect to the local device.If the Server Mode (second parameter) is set to anything other than ausManualAccept then no open request indication events will be dispatched.

This function returns zero if successful, or a negative return error code if an error occurred.

Note

The Default Server Connection Mode is AutomaticAccept. This function is used for Bluetooth devices operating in Bluetooth Security Mode 2.

QueryConnectionMode

Description

The following function is responsible for determining and displaying how incoming connections are handled.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the query.

Possible Return Values

(0) Successful

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2000) BTAUD_ERROR_INVALID_PARAMETER

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

API Call

AUD_Get_Server_Connection_Mode(BluetoothStackID, &ServerConnectionMode)

API Prototype

int BTPSAPI AUD_Get_Server_Connection_Mode(unsigned int BluetoothStackID, AUD_Server_Connection_Mode_t *ServerConnectionMode)

Description of API

The following function is responsible for retrieving the current Audio Manager Connection Mode.This function accepts as its first parameter the Bluetooth stack ID of the Bluetooth stack on which the server exists.The final parameter to this function is a pointer to a Server Connection Mode variable which will receive the current Server Connection Mode.

This function returns zero if successful, or a negative return error code if an error occurred.

Note

The Default Server Connection Mode is ausAutomaticAccept. This function is used for Bluetooth devices operating in Bluetooth Security Mode 2.

SendPassThroughCommand

Description

The following function is responsible for sending Remote Control Pass Through Commands.

Parameters

This function required one or two parameters, Commands (0 = Pause, 1 = Play, 2 = Stop, 3 = Vol. Up, 4 = Vol. Down, 5 = Specify Command with value parameter (the second parameter)).

Possible Return Values

(0) Successful

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-2000) BTAUD_ERROR_INVALID_PARAMETER

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2004) BTAUD_ERROR_INSUFFICIENT_RESOURCES

(-2023) BTAUD_ERROR_UNABLE_TO_SEND_REMOTE_CONTROL_COMMAND

(-2025) BTAUD_ERROR_REMOTE_DEVICE_NOT_CONNECTED

(-2026) BTAUD_ERROR_REMOTE_CONTROL_NOT_CONNECTED

(-2033) BTAUD_ERROR_REMOTE_CONTROL_ROLE_NOT_INITIALIZED

API Call

AUD_Send_Remote_Control_Command(BluetoothStackID, ConnectedBD_ADDR, &RemoteControlCommandData, 0)

AUD_Send_Remote_Control_Command(BluetoothStackID, ConnectedBD_ADDR, &RemoteControlCommandData, DEFAULT_PASS_THROUGH_COMMAND_TIMEOUT)

API Prototype

int BTPSAPI AUD_Send_Remote_Control_Command(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, AUD_Remote_Control_Command_Data_t *RemoteControlCommandData, unsigned long ResponseTimeout)

Description of API

The following function is responsible for sending the specified Remote Control Command to the remote device.This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the requested Audio Manager is present, followed by the Bluetooth device address of the device that is to receive the command, followed by the Remote Control information itself. The final parameter represents the Timeout value (in milliseconds) to wait for a response (confirmation) from the remote device.

This function returns a positive (non-zero) return value if successful which represents the internal Transaction ID of the Remote Control Command. This ID can be used to match responses (confirmations) with outgoing commands.This function returns a negative return error code if there was an error.

Note

Currently this function is ONLY applicable for local SNK devices (i.e. if there is a SRC connected and no SNK connected then this call will fail). Specifying a Timeout of zero will cause the Audio Manager to not track the command (and thus, not dispatch a response/confirmation event).

3 A3DP Sink Demo Guide

3.1 Demo Overview

The Assisted Advanced Audio Distribution Profile (A3DP) Sink allows a device to act as an Audio sink and can control and stream audio on an Audio source.

It is recommended that the user visits the kit setup Getting Started Guide for MSP430, Getting Started Guide for Tiva, Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Note

The same instructions can be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms. An external codec MUST be connected to the CC256x I2S/PCM interface to play audio except for the BT-MSPAUDSINK Reference Design board.

Running the Bluetooth Code Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as **MSP-EXP430F5438 USB - Serial Port(COM x), Tiva Virtual COM Port (COM x), XDS110 Class Application/User UART (COM x)** for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud (9600 for MSP430), 8, n, 1. Once connected, reset the device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands

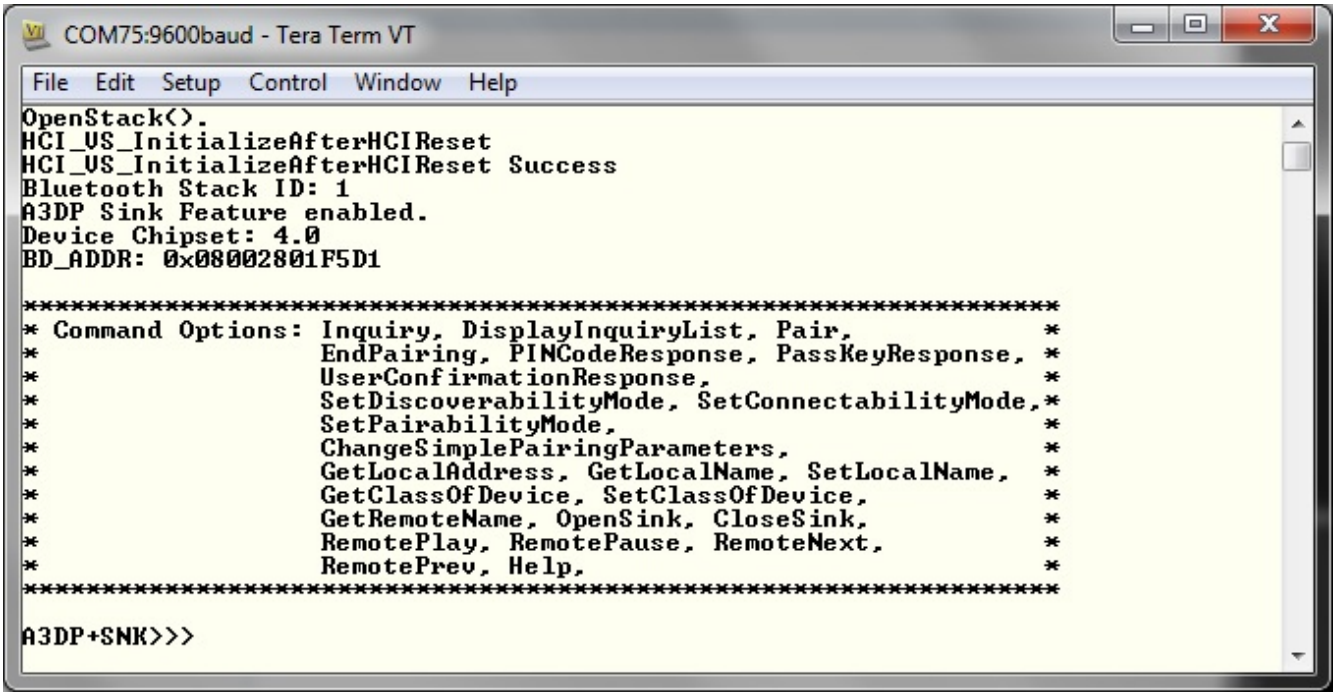

Figure 3-1. A3DP Sink Demo Start

3.2 Demo Application

This section provides a description of how to use the demo application to connect an audio source to it and communicate over Bluetooth.

Device 1 (Sink) setup on the demo application

1. Follow the steps in the running the bluetooth code section to setup the sink.
2. Set the name of the device. In the A3DP+SNK>>> prompt enter **SetLocalName** a3dpmsp430. Note that you can replace a3dpmsp430 with any other name.

3. If you are connecting to an Apple Device, you will need to set the class of device as well. In the A3DP+SNK>>> prompt enter **SetClassOfDevice 0x040424**. This set's the class of device to a headset.
4. Open the A3DP sink. In the A3DP+SNK>>> prompt enter **OpenSink**. This opens an A3DP endpoint.

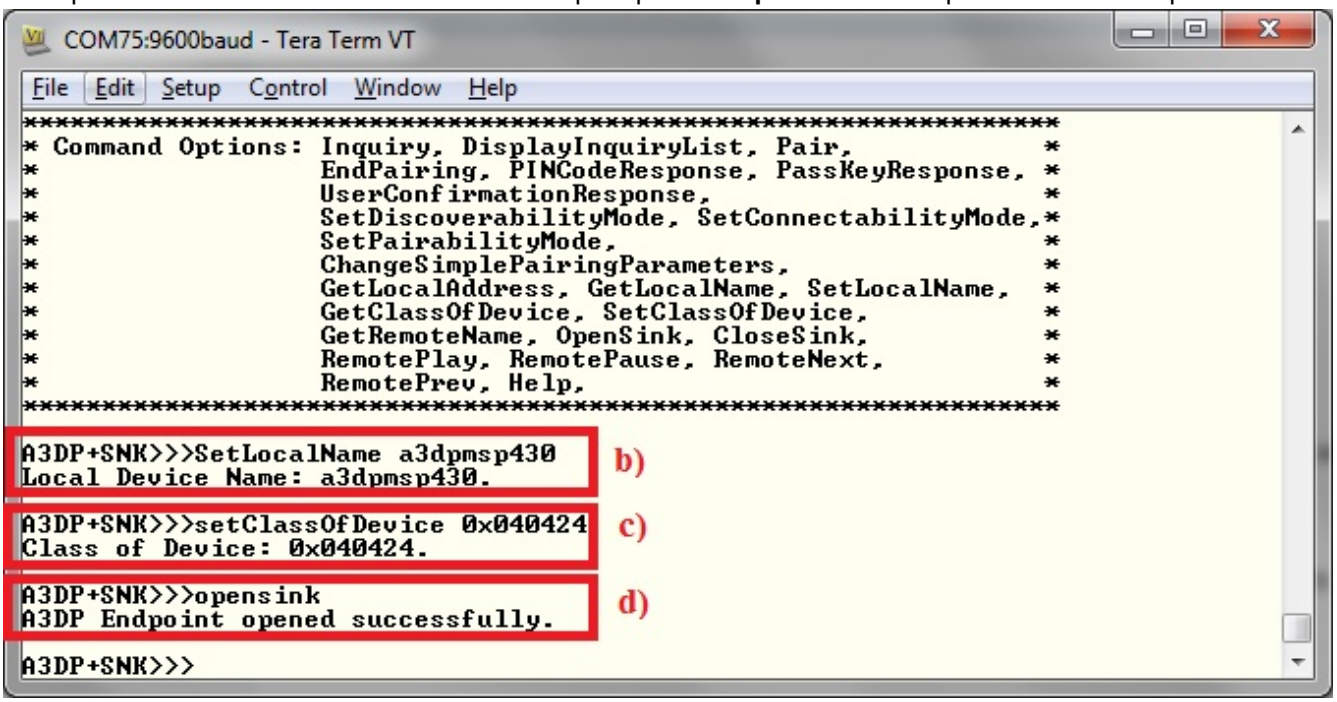

Figure 3-2. A3DP Sink Demo Set Terminal

Source Setup

The A3DP source can be any application that can transmit audio. For our example, we will use an android phone as the demo.

1. Open the bluetooth settings menu on the android phone(Settings->Bluetooth). We should get a menu like this.
2. Hit on search for devices. The phone should begin looking for other bluetooth devices.

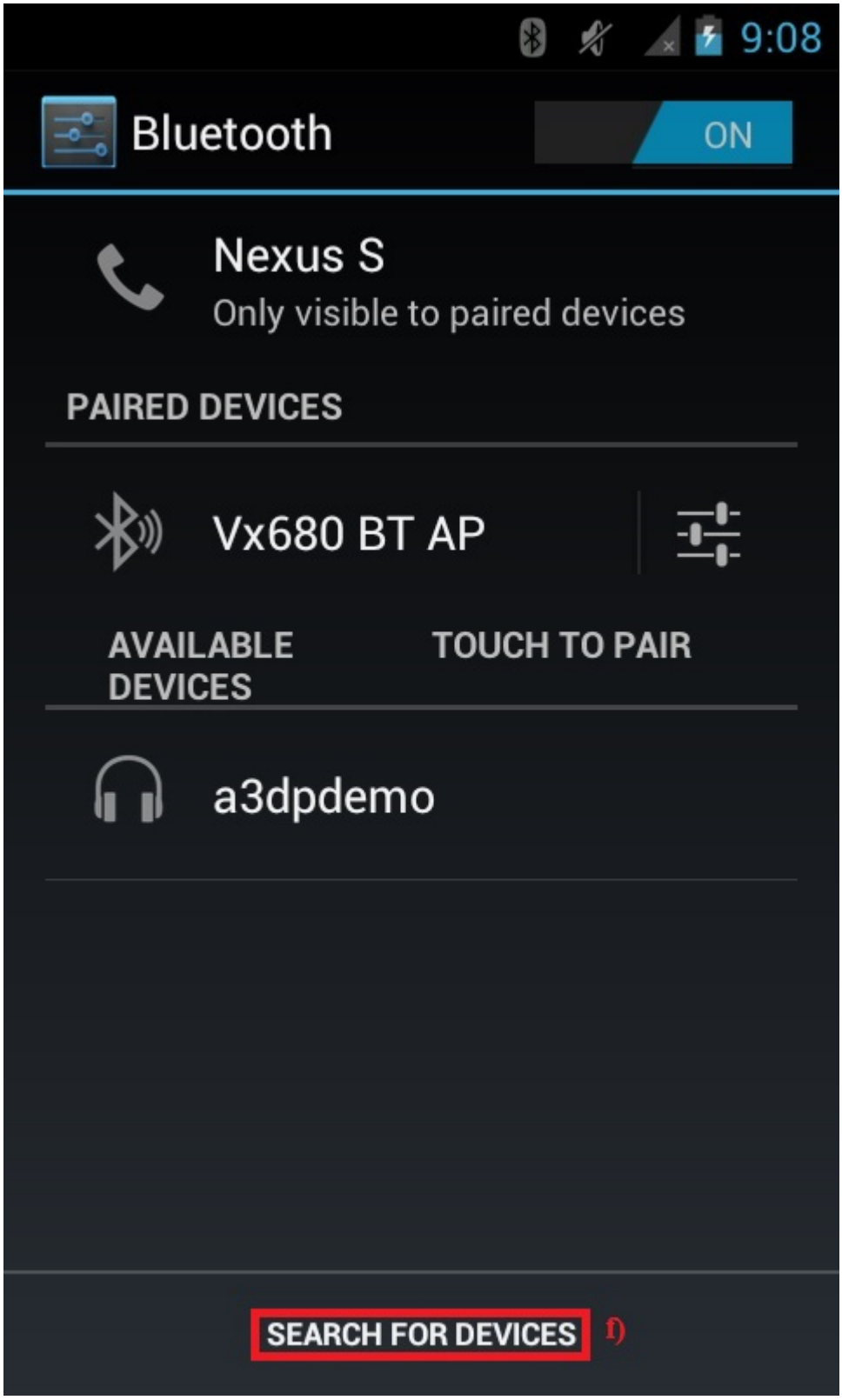

Figure 3-3. A3DP Sink Demo Bluetooth Settings

3. A A3DPDemo should appear like shown below in the picture. Click on the device to begin pairing.

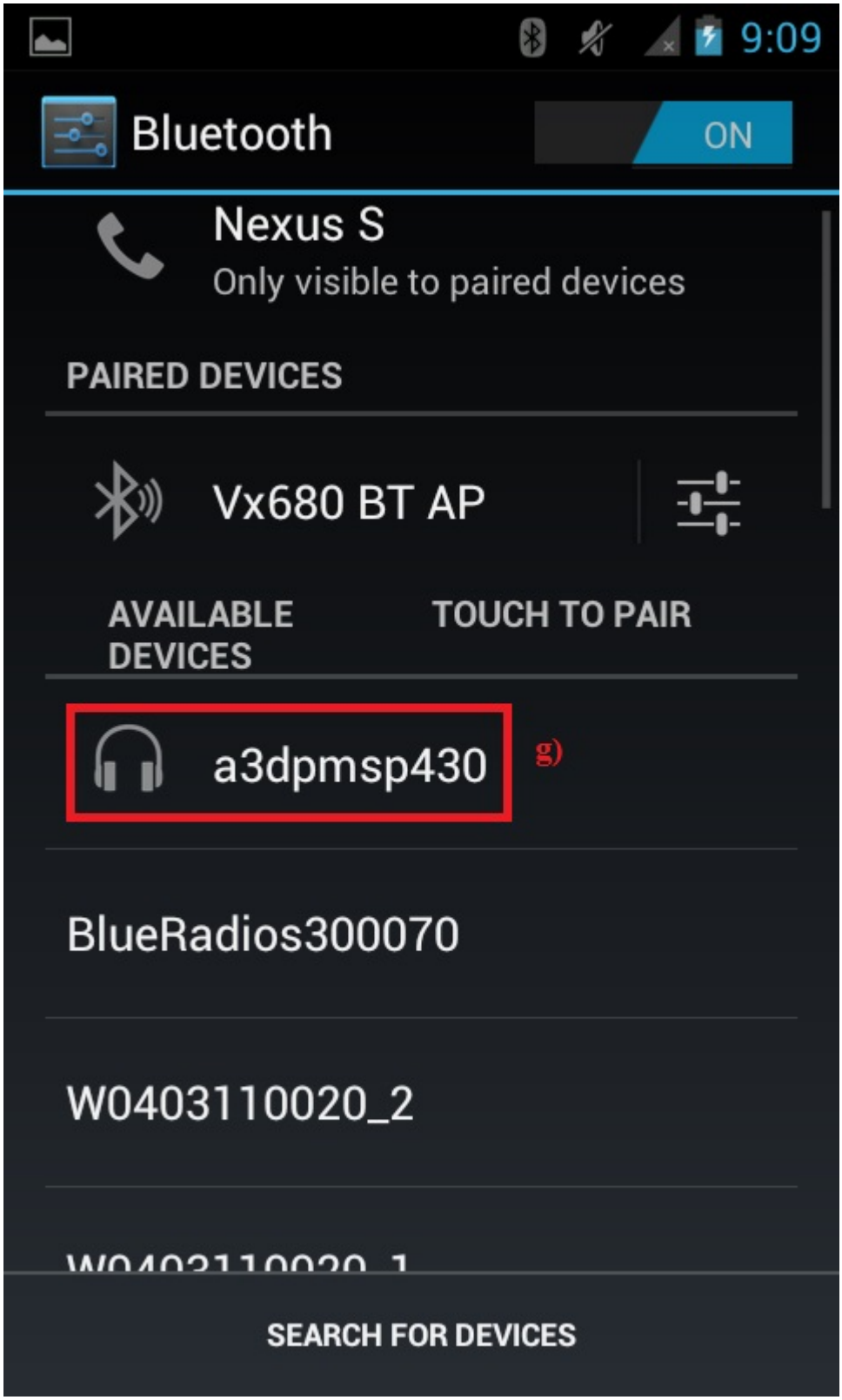

Figure 3-4. A3DP Sink Search for Devices

4. After the devices are paired, the device should show connected on the phone side and on the MSP430.

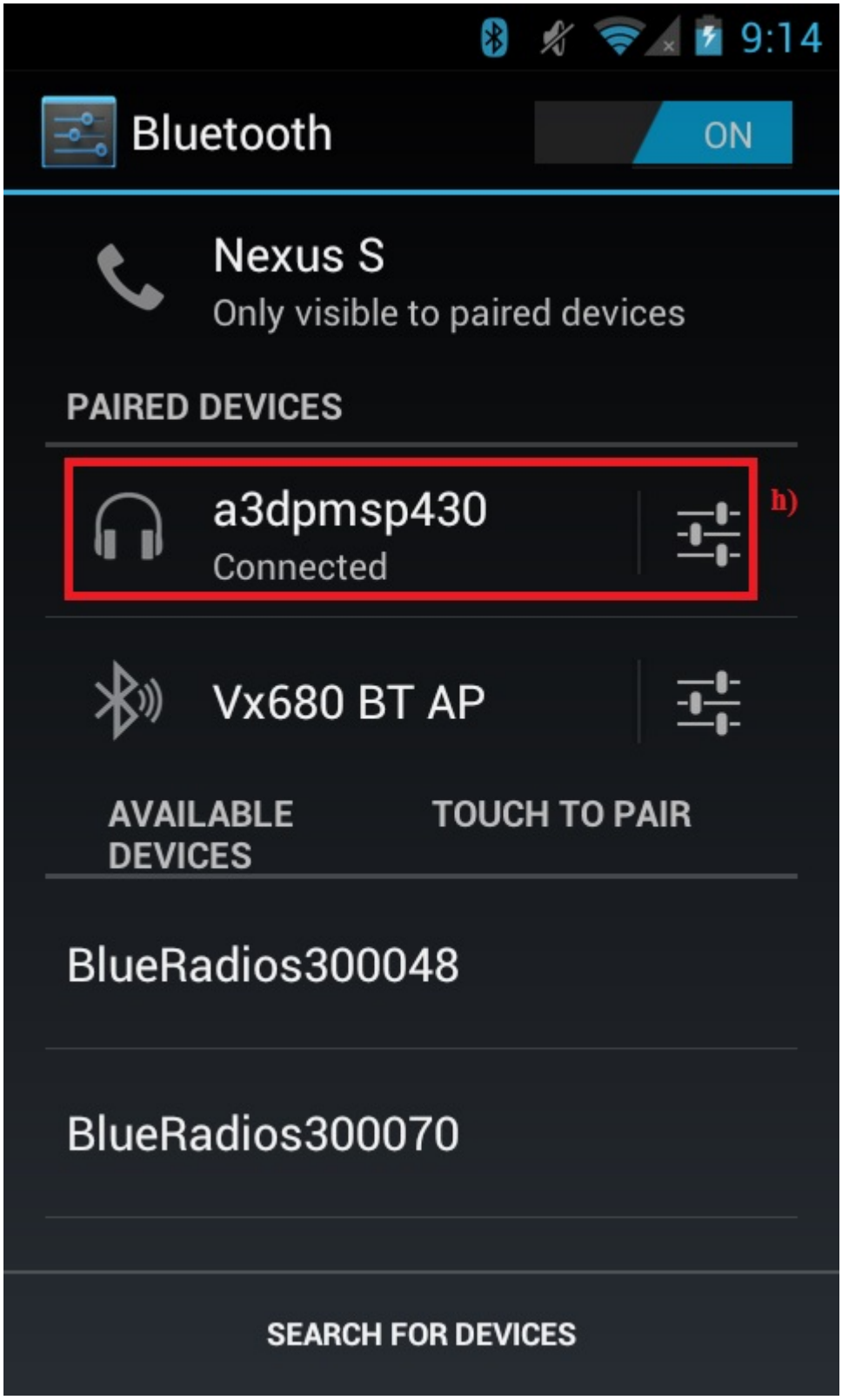

Figure 3-5. A3DP Sink Demo Paired Phone

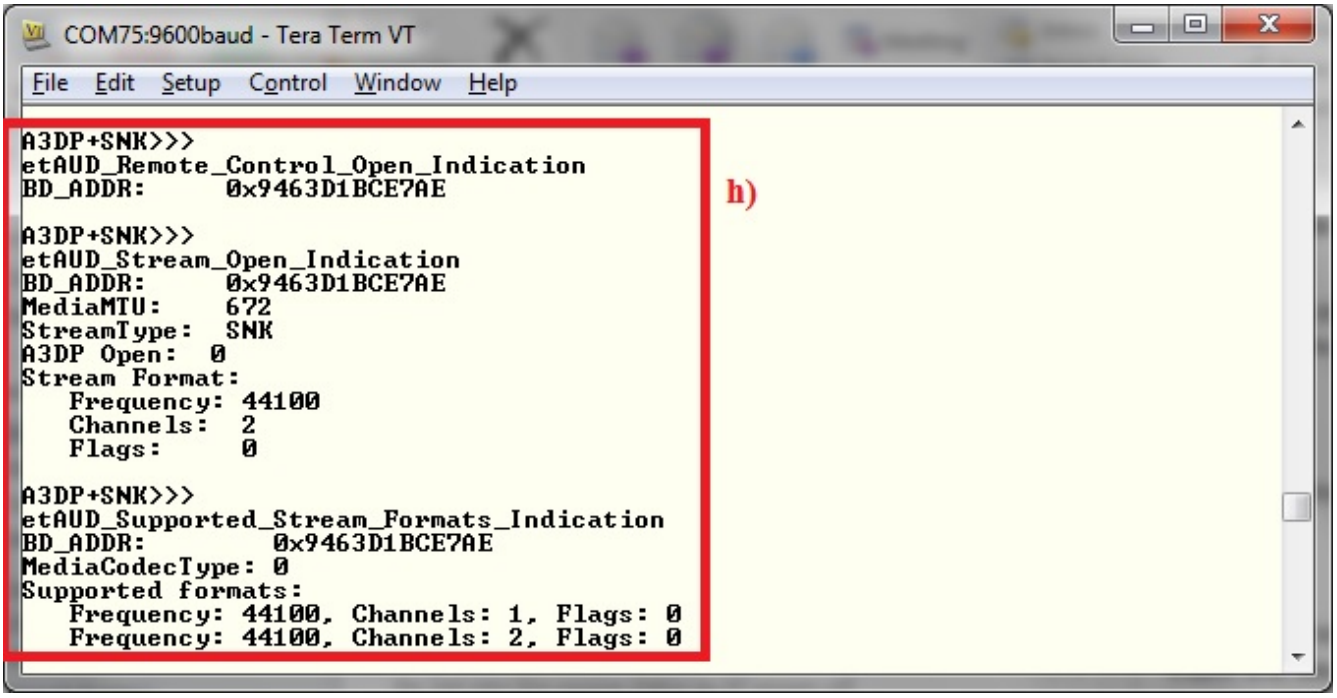

Figure 3-6. A3DP Connected MSP430

5. You can now control the audio of device 2 from the sink reference board board.

A3DP Commands

1. Open any music application, In the A3DP+SNK>>> prompt enter **RemotePlay**. A track starts playing on the device.
2. In the A3DP+SNK>>> prompt enter **RemotePause**. The current track is paused.

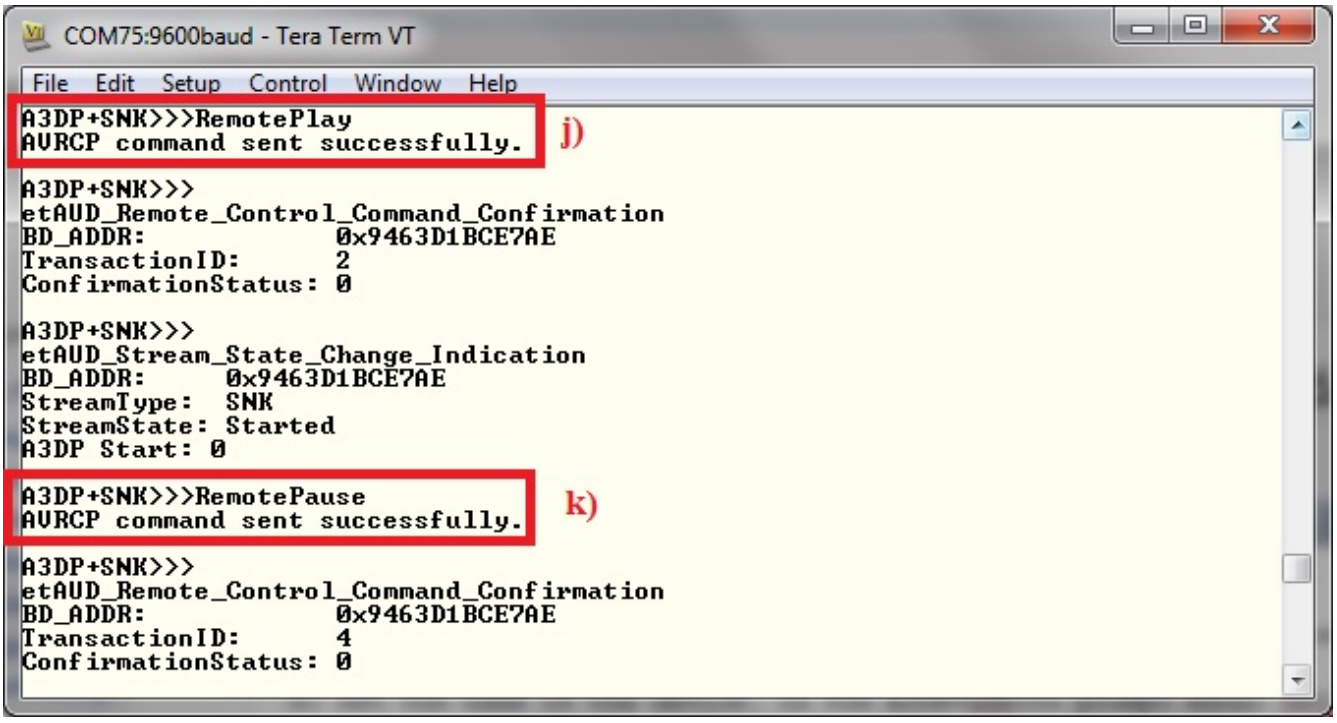

Figure 3-7. AD3P Sink Demo Play and Pause

3. In the A3DP+SNK>>> prompt enter **RemoteNext**. The next track is played.
4. In the A3DP+SNK>>> prompt enter **RemotePrev**. The previous track is played.

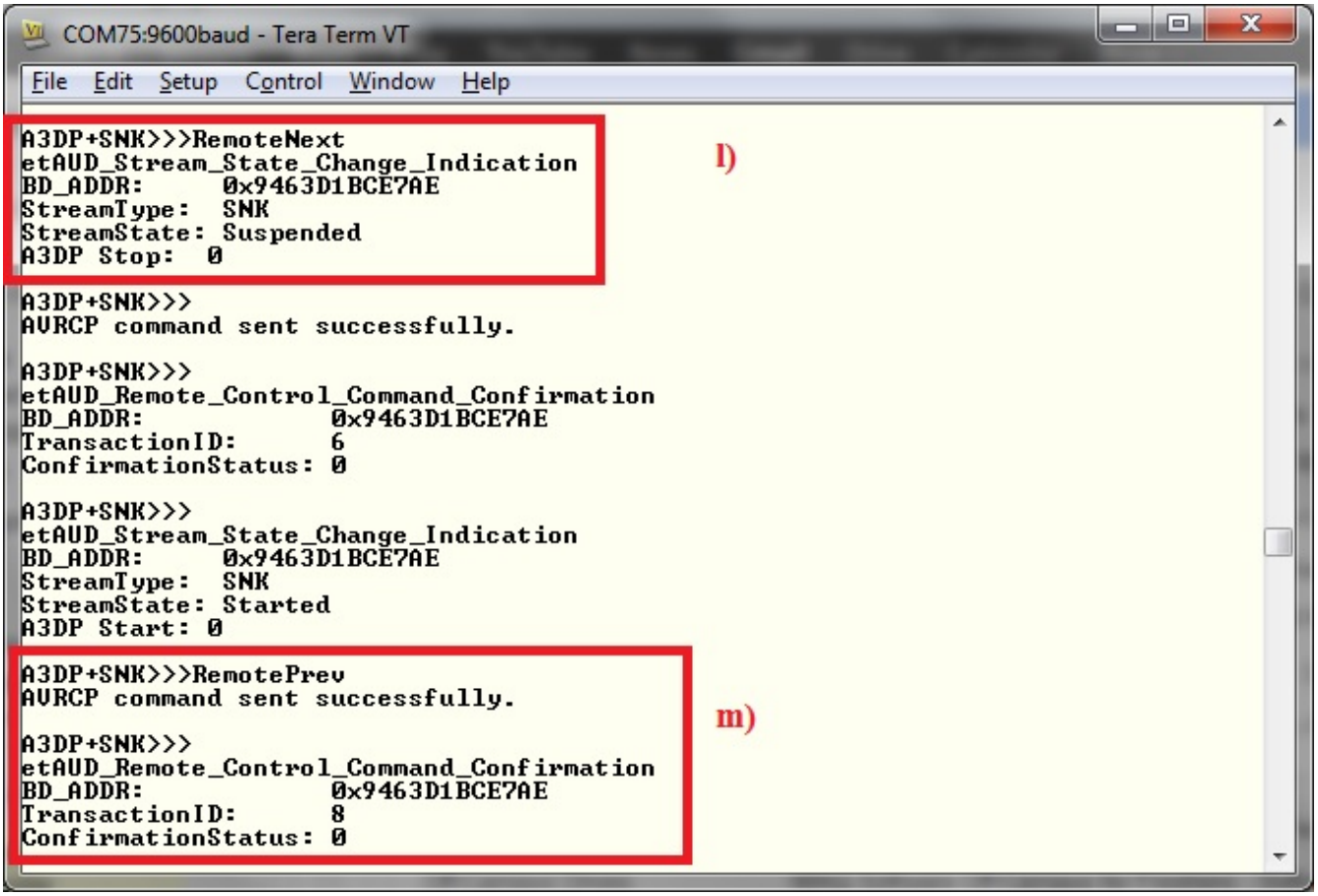

Figure 3-8. AD3P Sink Demo Prev and Next

3.3 Sink Reference Board Demo Application

To use the Assisted A2DP profile (A3DP) on the Sink Reference board, Make sure that the profile is flashed and the device is powered up and the switch is turned On.

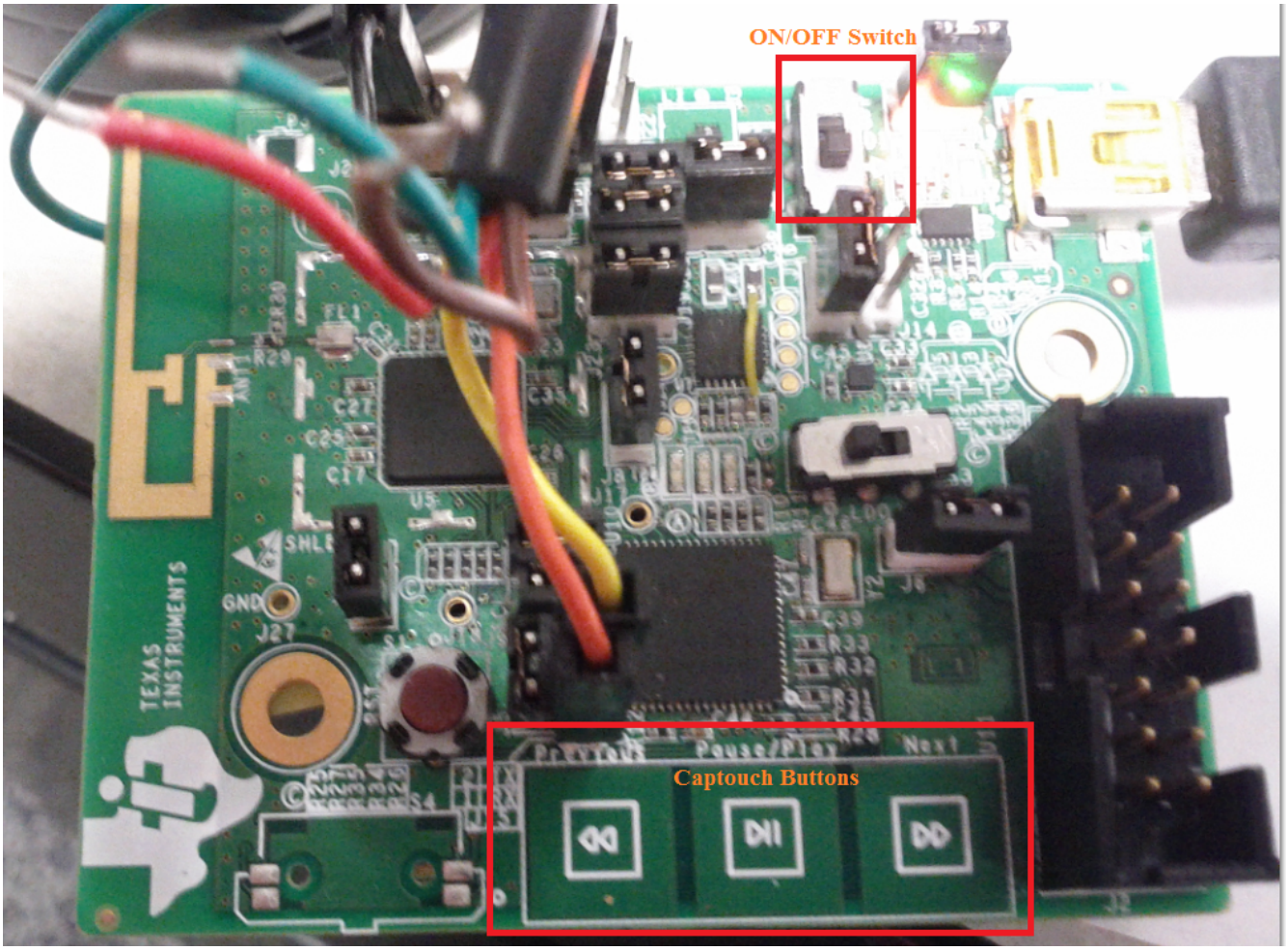

Figure 3-9. A3DP Board Setup and Sink Device

1. On the Phone Side, Make sure Bluetooth is turned on. Begin Searching for devices.

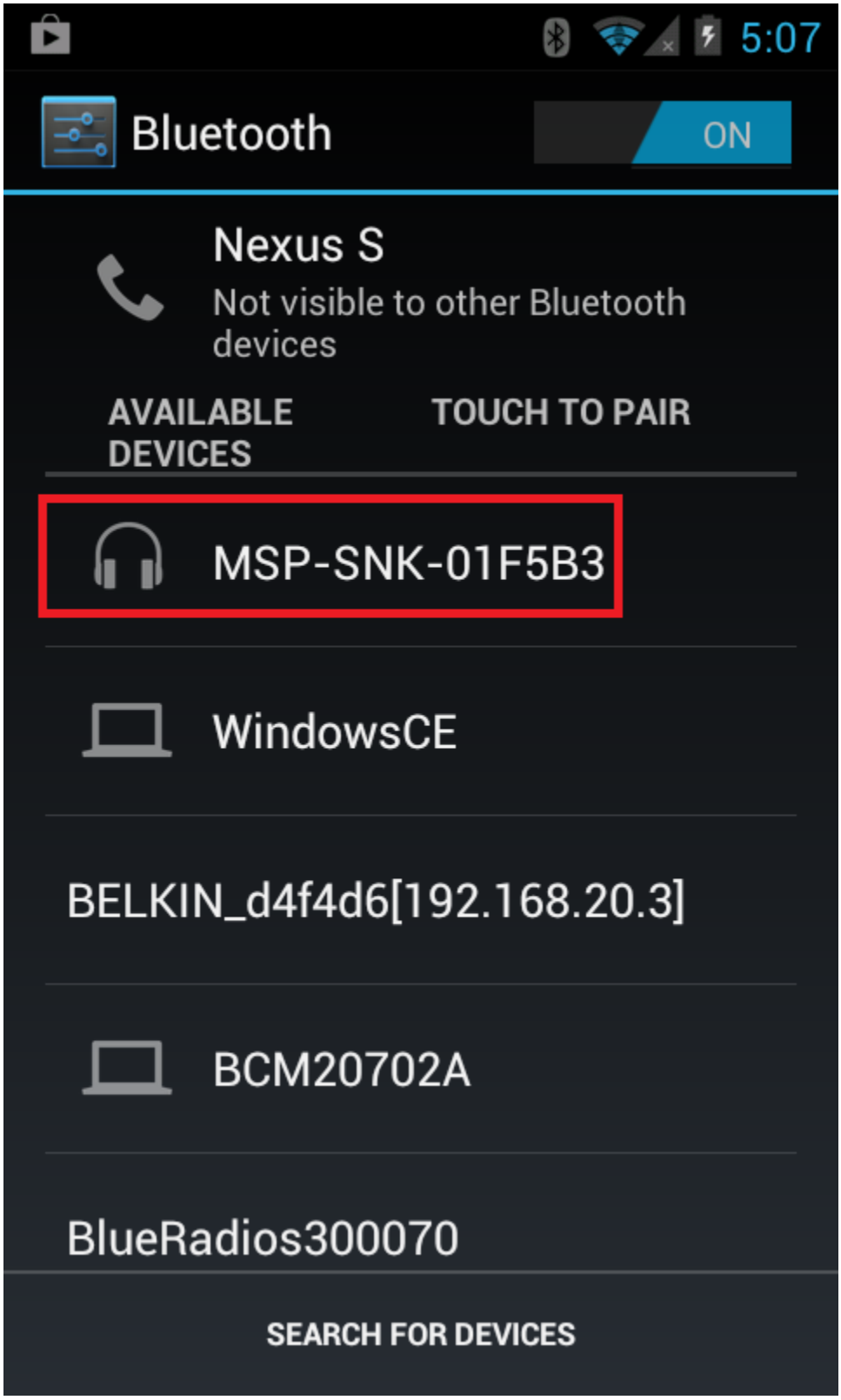

Figure 3-10. MSP Bluetooth Settings 1

2. Once discovery is done, connect and pair with the MSP-SNK-01F5B3.

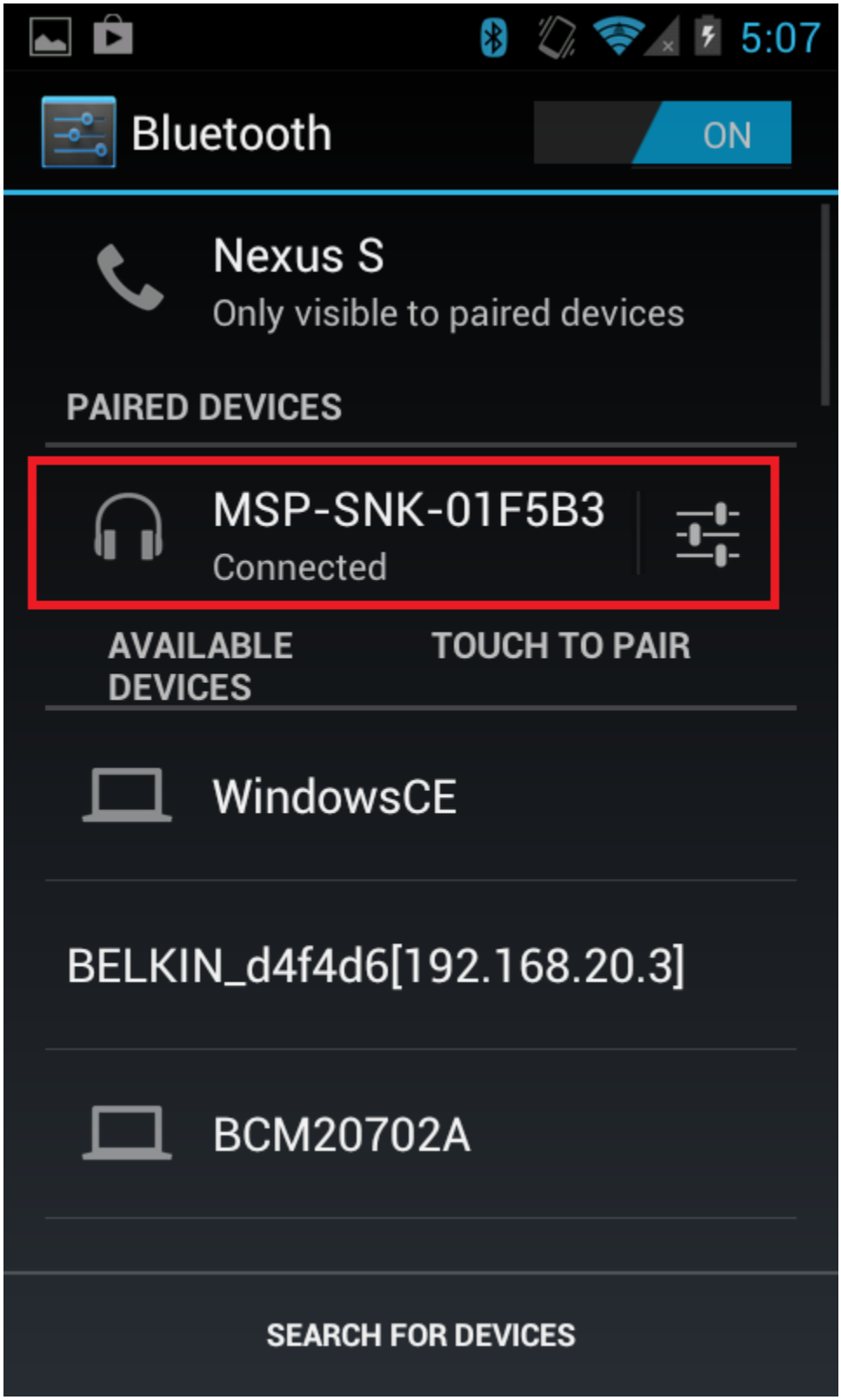

Figure 3-11. MSP Bluetooth Settings 2

3. Open up a music player. To play a song, press Play using the middle Captouch Button. To pause the song, Press the same button. To go to the Next song, press the >> on the Captouch Button and to go the Prev song, press the << on the Captouch Button.

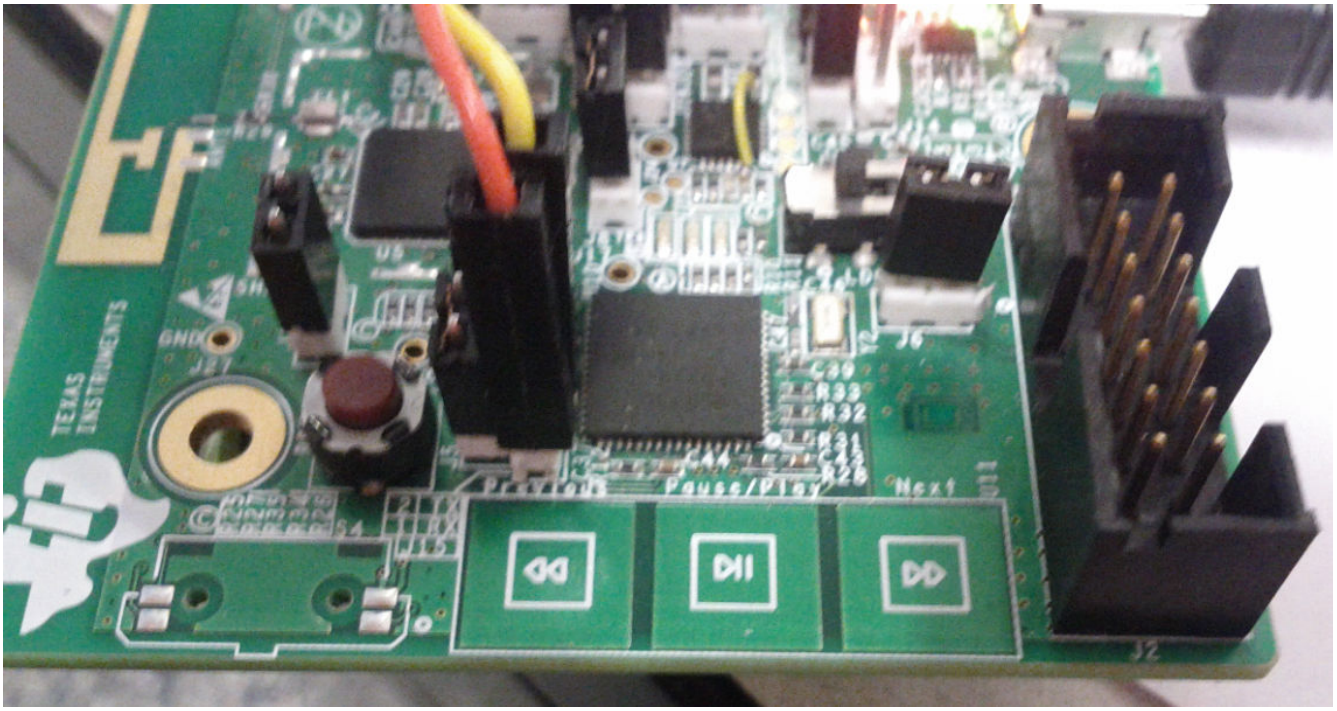

Figure 3-12. Board Setup of Sink Device

3.4 Multiple Source Demo

1. The A3DP Sink app allows for multiple sources to connect at the same time. With the Sink Reference board, connect and pair with an Android device as shown in the previous section.
2. Begin playing audio on the Android Device.
3. From another phone, begin searching for the sink device. We use an iPhone here.

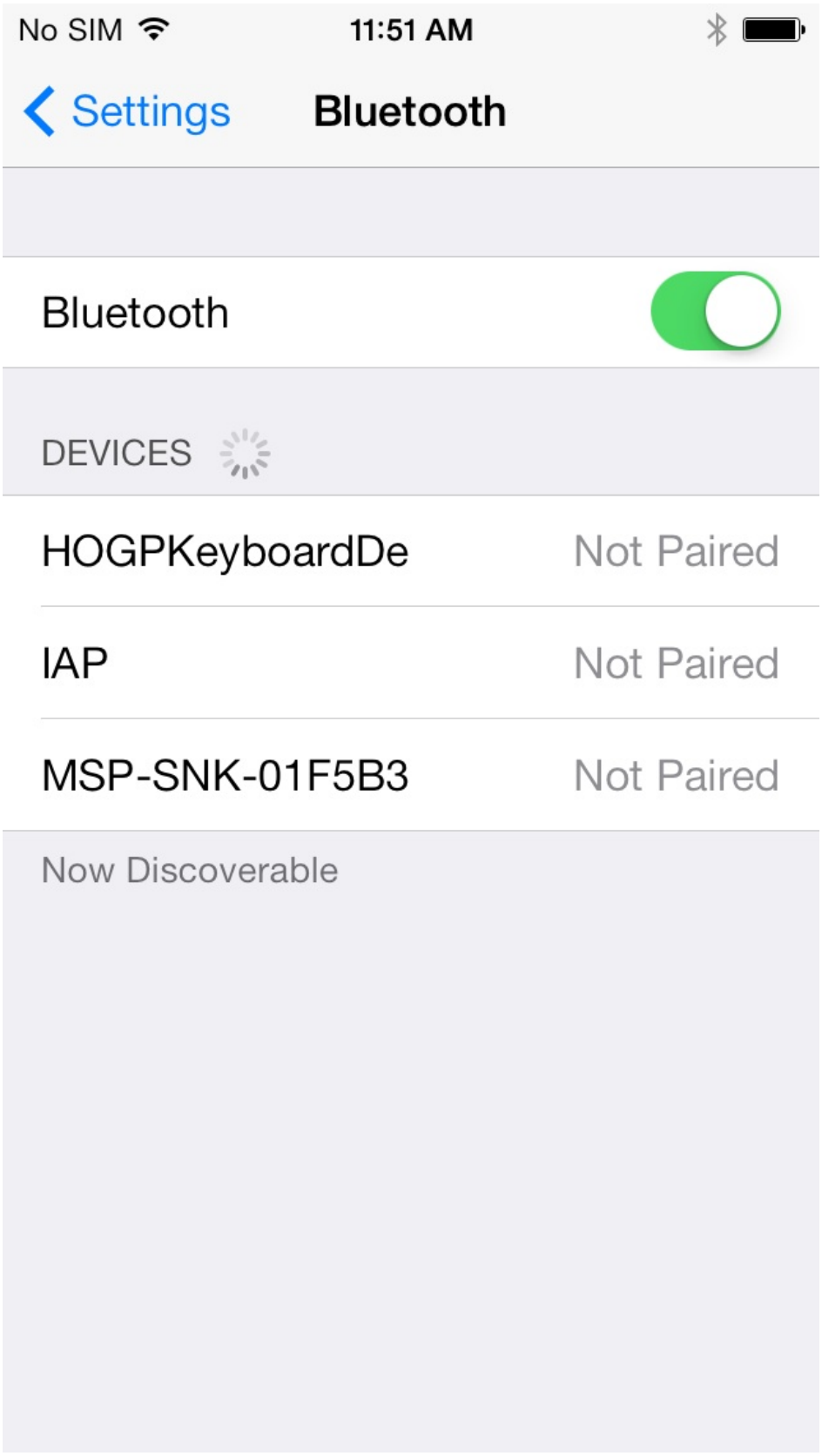

Figure 3-13. Searching for the sink device

4. Connect and pair with the sink device, the audio should pause on the Android device.

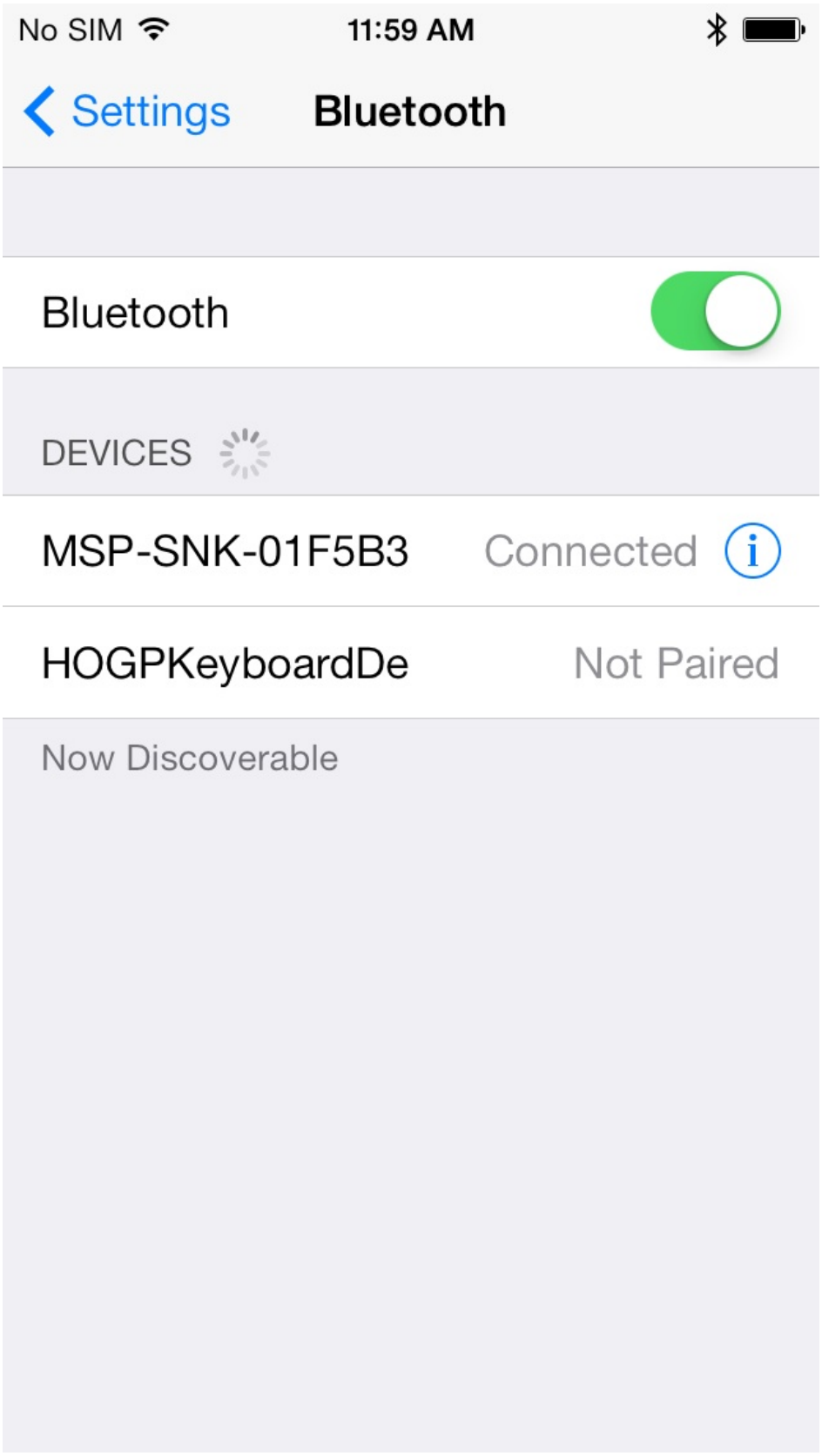

Figure 3-14. Connected to the sink device

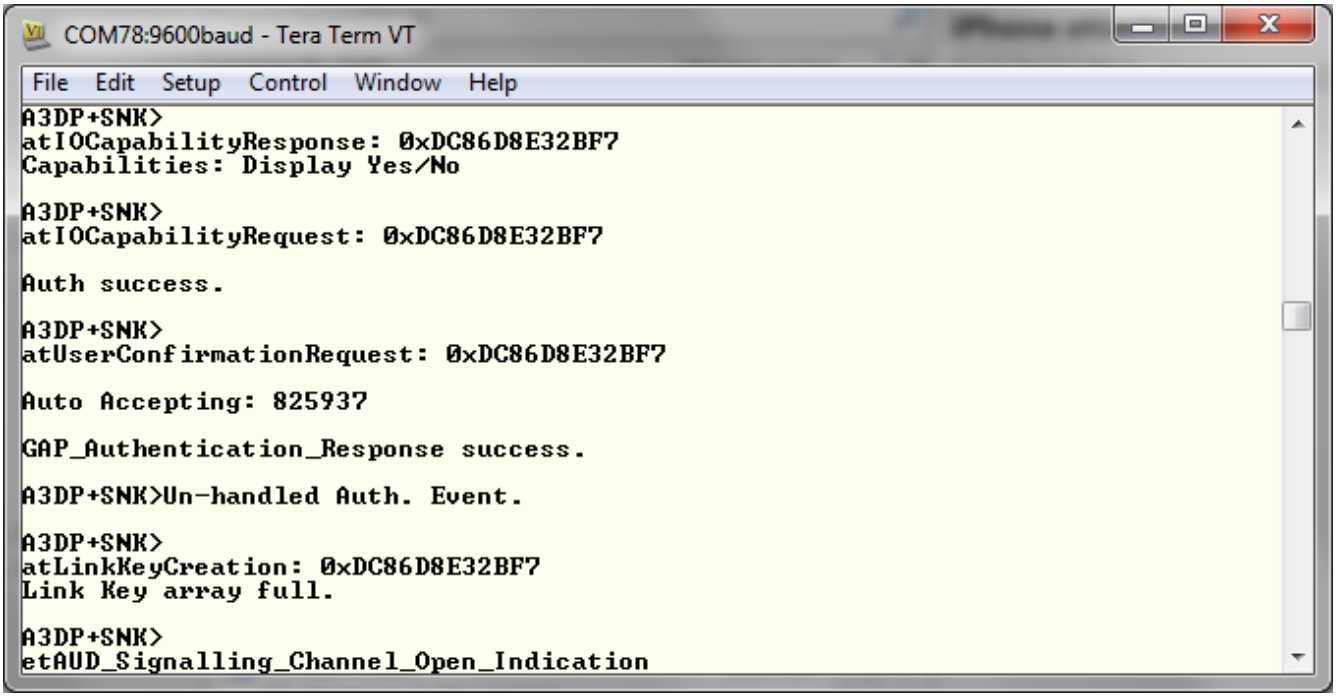

Figure 3-15. A3DP Terminal 1

5. Begin playing audio on the iPhone. It should play if you play audio from the android device, the iPhone will pause and the audio from the Android device will start.

3.5 Application Commands

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands

A3DP Profile Commands

OpenSink

Description

The following function is responsible for initializing AUD if necessary, and initializing the A3DP subsystem.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of OpenSink.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2005) BTAUD_ERROR_ALREADY_CONNECTED

(-2008) BTAUD_ERROR_STREAM_NOT_INITIALIZED

(-2010) BTAUD_ERROR_STREAM_ALREADY_CONNECTED

(-2013) BTAUD_ERROR_STREAM_IS_ACTIVE

(-2032) BTAUD_ERROR_STREAM_CONNECTED

API Call

AUD_Initialize(BluetoothStackID, NULL, &InitializationInfoSNK, AUD_Event_Callback, 0)

API Prototype

int BTPSAPI AUD_Initialize(unsigned int BluetoothStackID, AUD_Initialization_Info_t *SRCInitializationInfo, AUD_Initialization_Info_t *SNKInitializationInfo, AUD_Event_Callback_t EventCallback, unsigned long CallbackParameter)

Description of API

The following function is responsible for Registering an Audio Manager. Note that only one Audio Manager can be Registered for each Bluetooth stack. This function accepts the Bluetooth stack ID of the Bluetooth stack which this Server is to be associated with. The second parameter to this function is the Audio Manager Configuration Specification. The final two parameters specify the Audio Manager Event Callback function and Callback parameter, respectively, of the Audio Manager Event Callback that is to process any further events associated with this Audio Manager. This function returns zero if successful, or a negative return error code if an error occurred (see BTERRORS.H).

CloseSink

Description

The following function is responsible for cleaning up AUD and the A3DP stream, if the stream is opened and/or playing. **Parameters**

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the query.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2011) BTAUD_ERROR_STREAM_NOT_CONNECTED (-2014) BTAUD_ERROR_STREAM_IS_NOT_ACTIVE

API Call

AUD_Un_Initialize(BluetoothStackID);

API Prototype

int BTPSAPI AUD_Un_Initialize(unsigned int BluetoothStackID)

Description of API

The following function is responsible for Unregistering an Audio Manager (which was Registered by a successful call to either the AUD_Initialize() function. This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the Audio Manager was registered for. This function returns zero if successful, or a negative return error code if an error occurred see BTERRORS.H).

RemotePlay

Description

This function is responsible for handling an AVRCP Play command issued by the user.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the query.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2007) BTAUD_ERROR_UNABLE_TO_INITIALIZE_AVCTP

(-2008) BTAUD_ERROR_STREAM_NOT_INITIALIZED

(-2009) BTAUD_ERROR_UNABLE_TO_CONNECT_REMOTE_STREAM (-2011) BTAUD_ERROR_STREAM_NOT_CONNECTED

(-2014) BTAUD_ERROR_STREAM_IS_NOT_ACTIVE

(-2015) BTAUD_ERROR_STREAM_STATE_ALREADY_CURRENT

(-2016) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_STATE

(-2017) BTAUD_ERROR_STREAM_STATE_CHANGE_IN_PROGRESS

(-2018) BTAUD_ERROR_STREAM_FORMAT_CHANGE_IN_PROGRESS

(-2019) BTAUD_ERROR_UNSUPPORTED_FORMAT

(-2020) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_FORMAT

(-2021) BTAUD_ERROR_SAME_FORMAT

(-2022) BT_AUD_ERROR_RETRIEVING_SUPPORTED_FORMATS

(-2023) BTAUD_ERROR_UNABLE_TO_SEND_STREAM_DATA

(-2024) BTAUD_ERROR_UNABLE_TO_SEND_REMOTE_CONTROL_COMMAND

(-2026) BTAUD_ERROR_REMOTE_DEVICE_NOT_CONNECTED

(-2027) BTAUD_ERROR_REMOTE_CONTROL_NOT_CONNECTED

(-2028) BTAUD_ERROR_INVALID_REMOTE_CONTROL_DATA

(-2029) BTAUD_ERROR_REMOTE_CONTROL_ALREADY_CONNECTED

(-2030) BTAUD_ERROR_REMOTE_CONTROL_CONNECTION_IN_PROGRESS

(-2031) BTAUD_ERROR_REMOTE_CONTROL_NOT_INITIALIZED

API Call

SendRemoteControlCommand(rcPlay);

API Prototype

int SendRemoteControlCommand(RemoteControlCommand_t Command)

Description of API

The following function is used to send the specified remote control to the currently connected remote control device.

remote Pause

Description

This function is responsible for handling an AVRCP Pause command issued by the user.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the query.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2007) BTAUD_ERROR_UNABLE_TO_INITIALIZE_AVCTP

(-2008) BTAUD_ERROR_STREAM_NOT_INITIALIZED

(-2009) BTAUD_ERROR_UNABLE_TO_CONNECT_REMOTE_STREAM

(-2011) BTAUD_ERROR_STREAM_NOT_CONNECTED

(-2014) BTAUD_ERROR_STREAM_IS_NOT_ACTIVE

(-2015) BTAUD_ERROR_STREAM_STATE_ALREADY_CURRENT

(-2016) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_STATE

(-2017) BTAUD_ERROR_STREAM_STATE_CHANGE_IN_PROGRESS

(-2018) BTAUD_ERROR_STREAM_FORMAT_CHANGE_IN_PROGRESS

(-2019) BTAUD_ERROR_UNSUPPORTED_FORMAT

(-2020) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_FORMAT

(-2021) BTAUD_ERROR_SAME_FORMAT

(-2022) BT_AUD_ERROR_RETRIEVING_SUPPORTED_FORMATS

(-2023) BTAUD_ERROR_UNABLE_TO_SEND_STREAM_DATA

(-2024) BTAUD_ERROR_UNABLE_TO_SEND_REMOTE_CONTROL_COMMAND

(-2026) BTAUD_ERROR_REMOTE_DEVICE_NOT_CONNECTED

(-2027) BTAUD_ERROR_REMOTE_CONTROL_NOT_CONNECTED

(-2028) BTAUD_ERROR_INVALID_REMOTE_CONTROL_DATA

(-2029) BTAUD_ERROR_REMOTE_CONTROL_ALREADY_CONNECTED

(-2030) BTAUD_ERROR_REMOTE_CONTROL_CONNECTION_IN_PROGRESS

(-2031) BTAUD_ERROR_REMOTE_CONTROL_NOT_INITIALIZED

API Call

SendRemoteControlCommand(rcPause);

API Prototype

int SendRemoteControlCommand(RemoteControlCommand_t Command)

Description of API

The following function is used to send the specified remote control to the currently connected remote control device.

RemoteNext

Description

This function is responsible for handling an AVRCP Next command issued by the user.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the query.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2007) BTAUD_ERROR_UNABLE_TO_INITIALIZE_AVCTP

(-2008) BTAUD_ERROR_STREAM_NOT_INITIALIZED

(-2009) BTAUD_ERROR_UNABLE_TO_CONNECT_REMOTE_STREAM

(-2011) BTAUD_ERROR_STREAM_NOT_CONNECTED

(-2014) BTAUD_ERROR_STREAM_IS_NOT_ACTIVE

(-2015) BTAUD_ERROR_STREAM_STATE_ALREADY_CURRENT

(-2016) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_STATE

(-2017) BTAUD_ERROR_STREAM_STATE_CHANGE_IN_PROGRESS

(-2018) BTAUD_ERROR_STREAM_FORMAT_CHANGE_IN_PROGRESS

(-2019) BTAUD_ERROR_UNSUPPORTED_FORMAT

(-2020) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_FORMAT

(-2021) BTAUD_ERROR_SAME_FORMAT

(-2022) BT_AUD_ERROR_RETRIEVING_SUPPORTED_FORMATS

(-2023) BTAUD_ERROR_UNABLE_TO_SEND_STREAM_DATA

(-2024) BTAUD_ERROR_UNABLE_TO_SEND_REMOTE_CONTROL_COMMAND

(-2026) BTAUD_ERROR_REMOTE_DEVICE_NOT_CONNECTED

(-2027) BTAUD_ERROR_REMOTE_CONTROL_NOT_CONNECTED

(-2028) BTAUD_ERROR_INVALID_REMOTE_CONTROL_DATA

(-2029) BTAUD_ERROR_REMOTE_CONTROL_ALREADY_CONNECTED

(-2030) BTAUD_ERROR_REMOTE_CONTROL_CONNECTION_IN_PROGRESS

(-2031) BTAUD_ERROR_REMOTE_CONTROL_NOT_INITIALIZED

API Call

SendRemoteControlCommand(rcNext);

API Prototype

int SendRemoteControlCommand(RemoteControlCommand_t Command)

Description of API

The following function is used to send the specified remote control to the currently connected remote control device.

RemotePrev

Description

This function is responsible for handling an AVRCP Back command issued by the user.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the query.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2007) BTAUD_ERROR_UNABLE_TO_INITIALIZE_AVCTP

(-2008) BTAUD_ERROR_STREAM_NOT_INITIALIZED

(-2009) BTAUD_ERROR_UNABLE_TO_CONNECT_REMOTE_STREAM

(-2011) BTAUD_ERROR_STREAM_NOT_CONNECTED

(-2014) BTAUD_ERROR_STREAM_IS_NOT_ACTIVE

(-2015) BTAUD_ERROR_STREAM_STATE_ALREADY_CURRENT

(-2016) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_STATE

(-2017) BTAUD_ERROR_STREAM_STATE_CHANGE_IN_PROGRESS

(-2018) BTAUD_ERROR_STREAM_FORMAT_CHANGE_IN_PROGRESS

(-2019) BTAUD_ERROR_UNSUPPORTED_FORMAT

(-2020) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_FORMAT

(-2021) BTAUD_ERROR_SAME_FORMAT

(-2022) BT_AUD_ERROR_RETRIEVING_SUPPORTED_FORMATS

(-2023) BTAUD_ERROR_UNABLE_TO_SEND_STREAM_DATA

(-2024) BTAUD_ERROR_UNABLE_TO_SEND_REMOTE_CONTROL_COMMAND

(-2026) BTAUD_ERROR_REMOTE_DEVICE_NOT_CONNECTED

(-2027) BTAUD_ERROR_REMOTE_CONTROL_NOT_CONNECTED

(-2028) BTAUD_ERROR_INVALID_REMOTE_CONTROL_DATA

(-2029) BTAUD_ERROR_REMOTE_CONTROL_ALREADY_CONNECTED

(-2030) BTAUD_ERROR_REMOTE_CONTROL_CONNECTION_IN_PROGRESS

(-2031) BTAUD_ERROR_REMOTE_CONTROL_NOT_INITIALIZED

API Call

SendRemoteControlCommand(rcBack);

API Prototype

int SendRemoteControlCommand(RemoteControlCommand_t Command)

Description of API

The following function is used to send the specified remote control to the currently connected remote control device.

4 A3DP Source Demo Guide

4.1 Demo Overview

The Assisted Advanced Audio Distribution Profile Source allows a device to act as an Audio Source and stream audio to an Audio Sink.

It is recommended that the user visits the kit setup Getting Started Guide for MSP430, Getting Started Guide for Tiva, Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Note

An external codec MUST be connected to the CC256x I2S/PCM interface to record audio, except for the BT-MSPAUDSOURCE Reference Design board. The same instructions can be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as MSP-EXP430F5438 USB - Serial Port (COM x), Tiva Virtual COM Port (COM x), XDS110 Class Application/User UART (COM x) for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud (9600 for MSP430), 8, n, 1. Once connected, reset the device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

Note

In IAR make sure that the correct version of the demo (Line-IN) for your board is flashed.

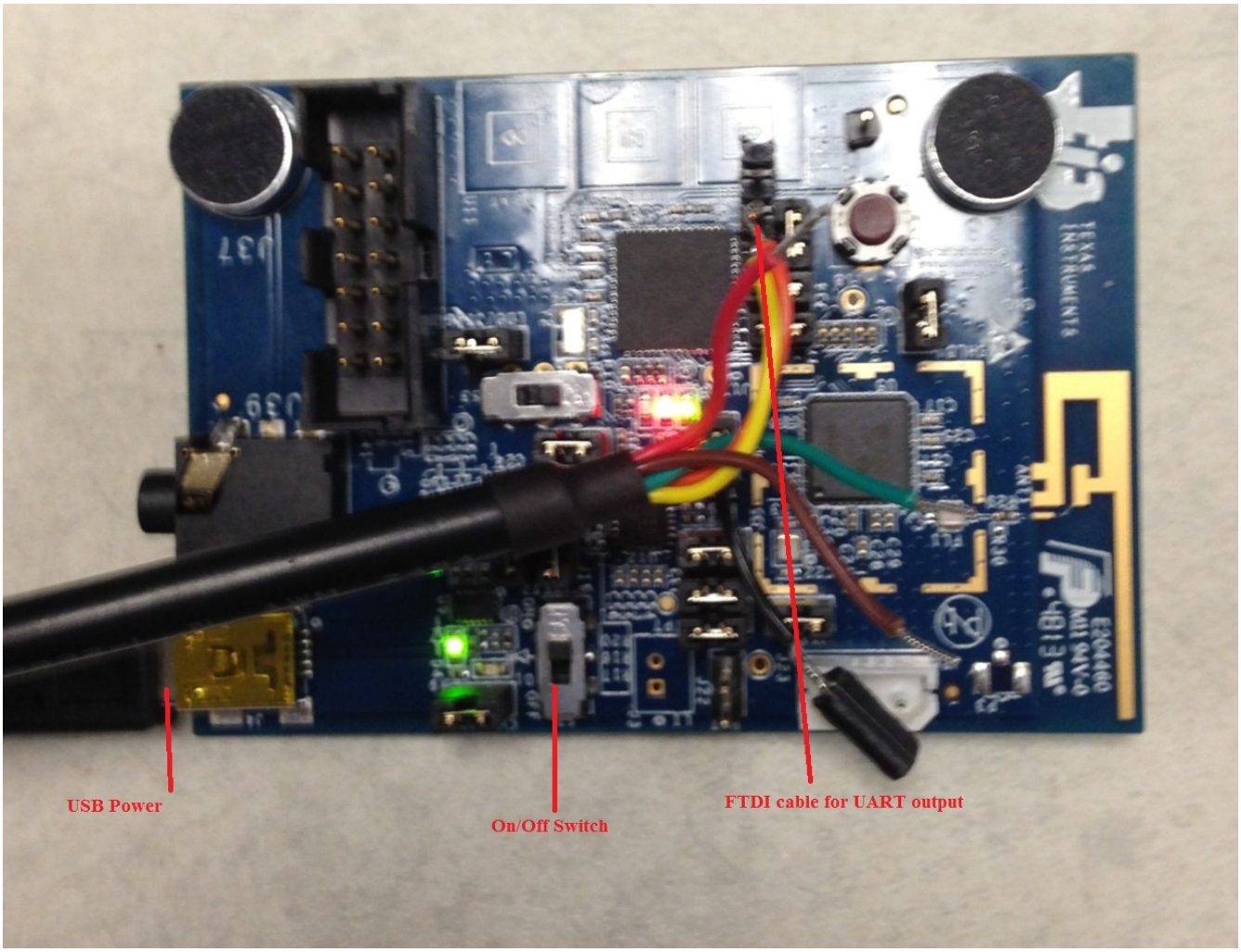

Figure 4-1. A3DP Source Demo Hardware

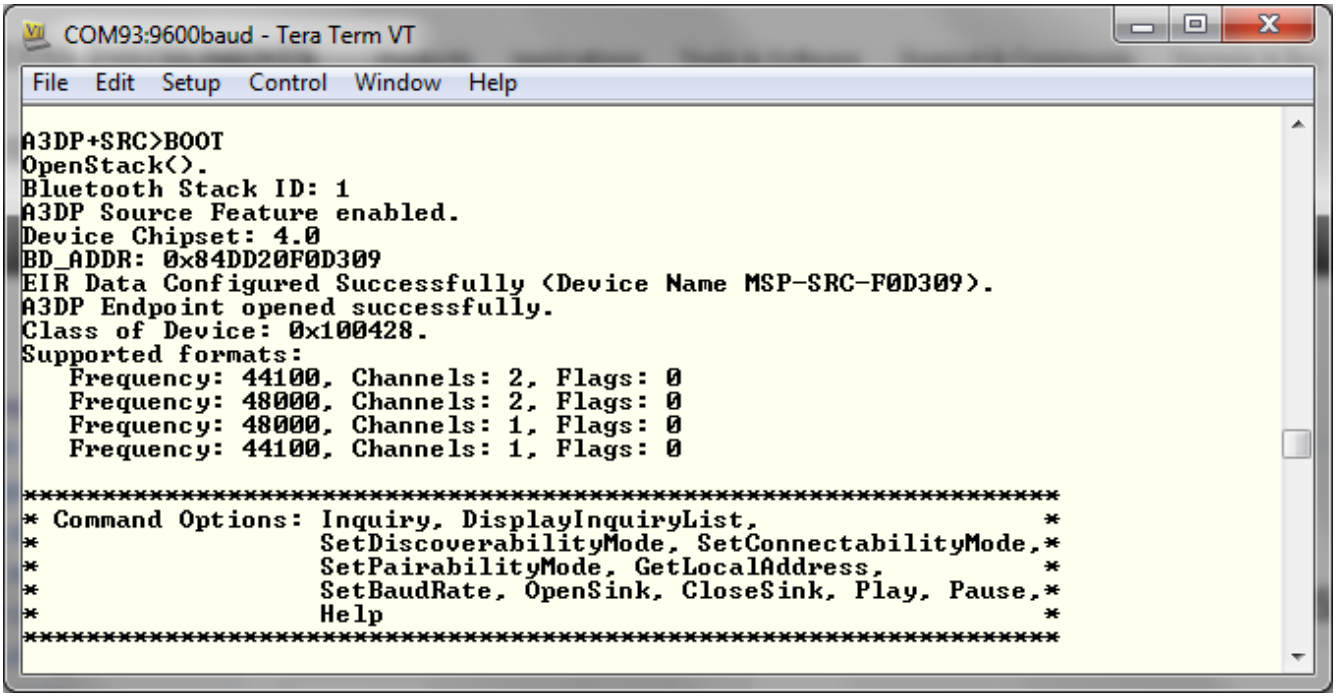

Figure 4-2. A3DP Source Demo Start Terminal

4.2 Demo Application

This section provides a description of how to use the demo application to connect an audio sink to it and communicate over Bluetooth.

Connecting with a Bluetooth Speaker
Device 2 (Sink) setup

1. Before we turn on the Source Board, we make sure that the Sink is ready to connect. In our example, we use a Bluetooth Speaker.

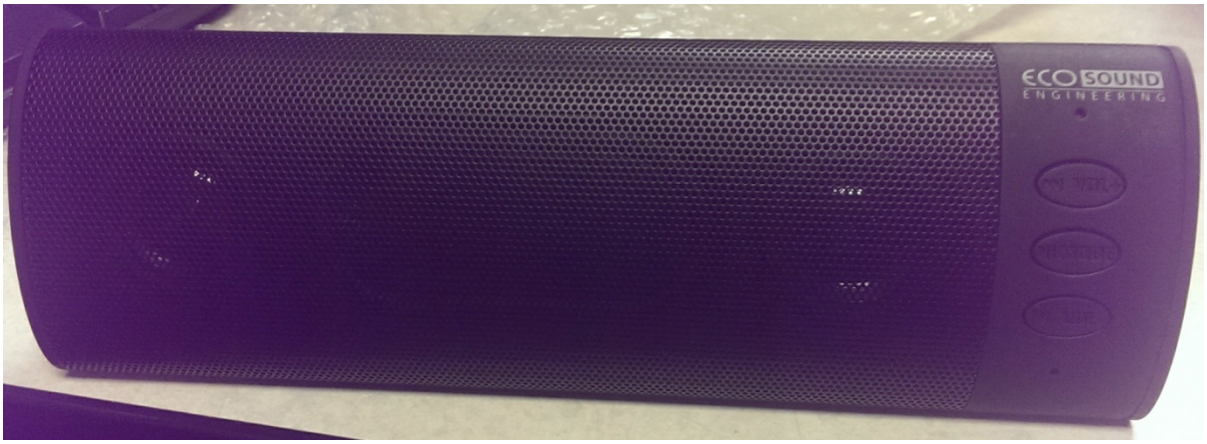

Figure 4-3. A3DP Source Setup

2. Turn on the speaker using the On/Off Switch. You should hear an audible beep.
3. Hold the mode button until you hear an audible beep and the blue led is flashing. The device is now discoverable and ready to be paired.

Device 1 (Source) setup on the demo application

1. Make sure that a microphone or similar device is connected to J39. Once the FTDI/microUSB cable is connected, press reset and you should see output similar to the Terminal output above.

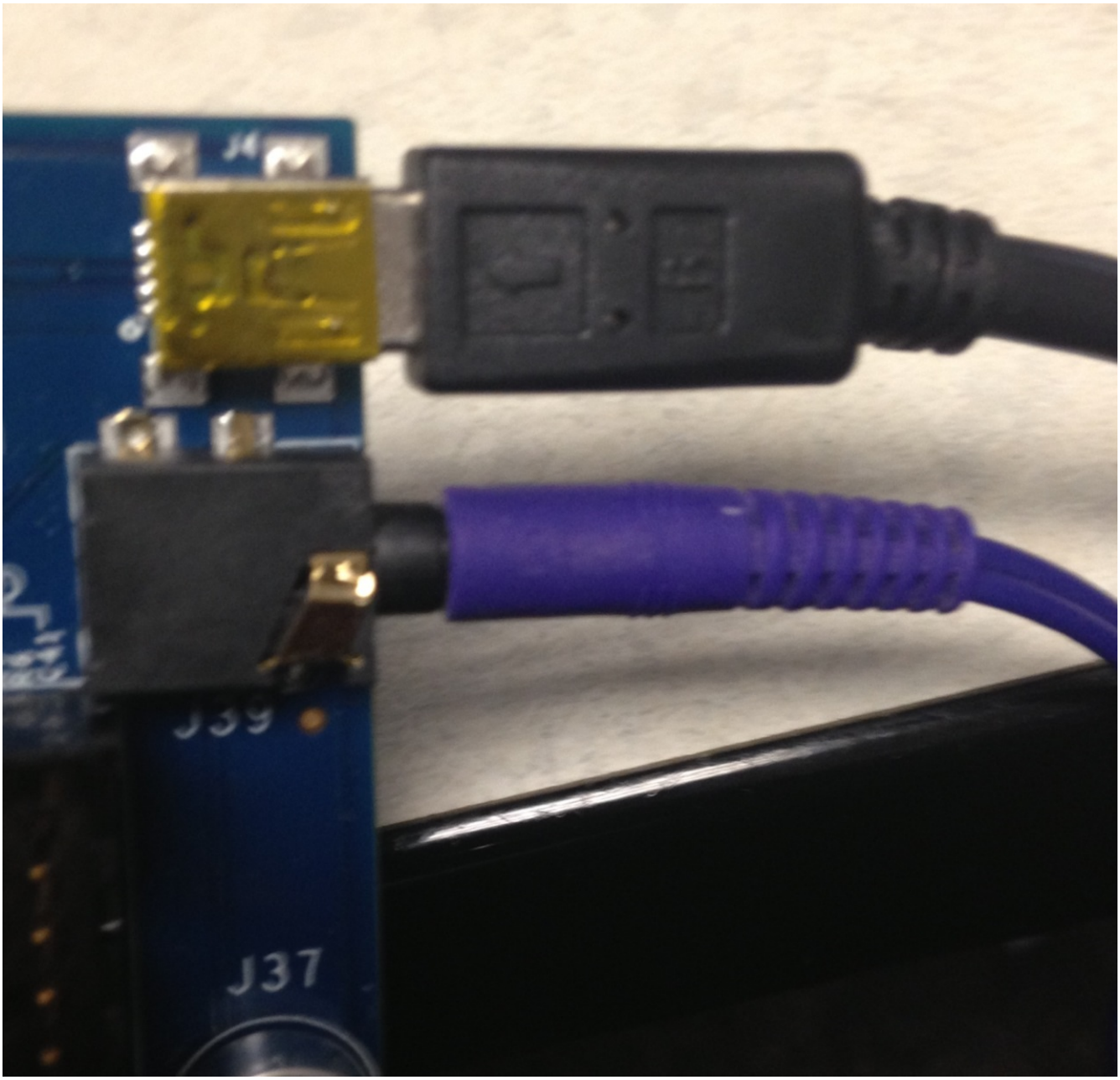

Figure 4-4. A3DP Source Setup Connections

2. The Source device will begin auto connecting with any discoverable & connectable sink device and filter out other devices.

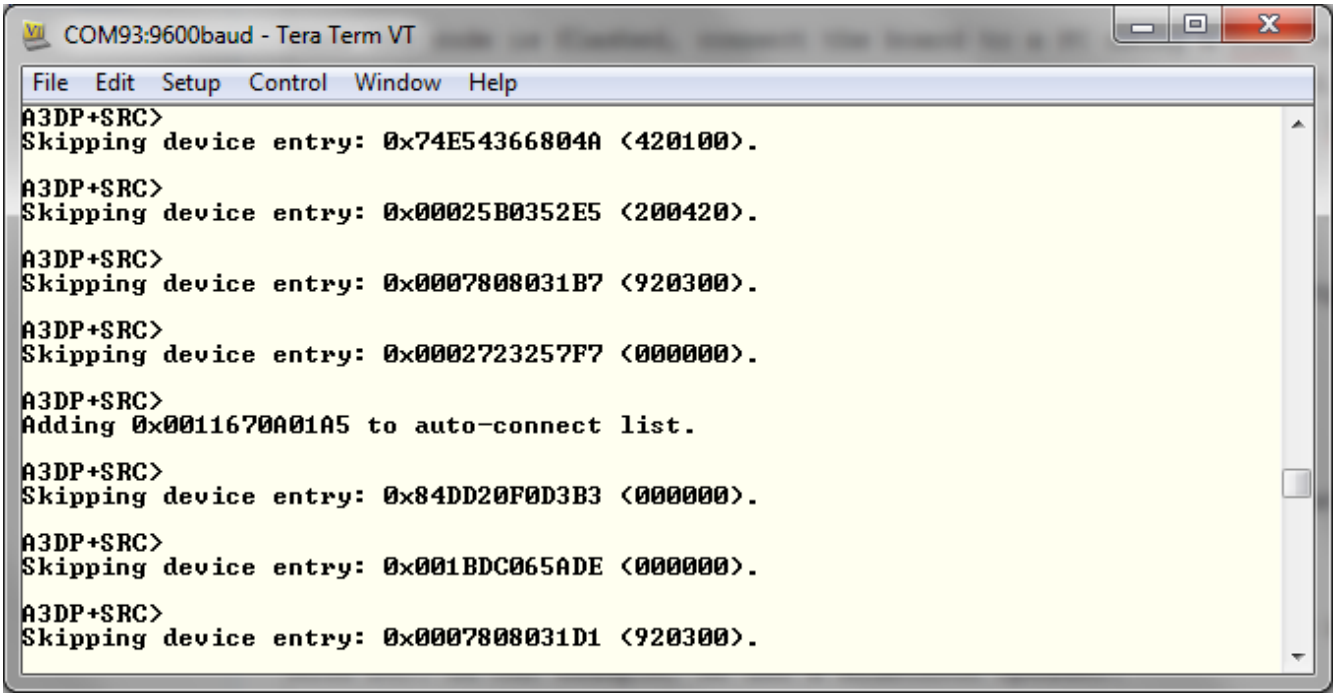

Figure 4-5. A3DP Source Demo Device List

Note

The device will automatically begin an inquiry/open a sink. If you need to do this manually, follow the steps and reference the Terminal Output below.

a. Type Inquiry to begin scanning for nearby devices.
b. Type OpenSink <Inquiry Number> to open a sink channel to a Bluetooth device found during the previous inquiry.

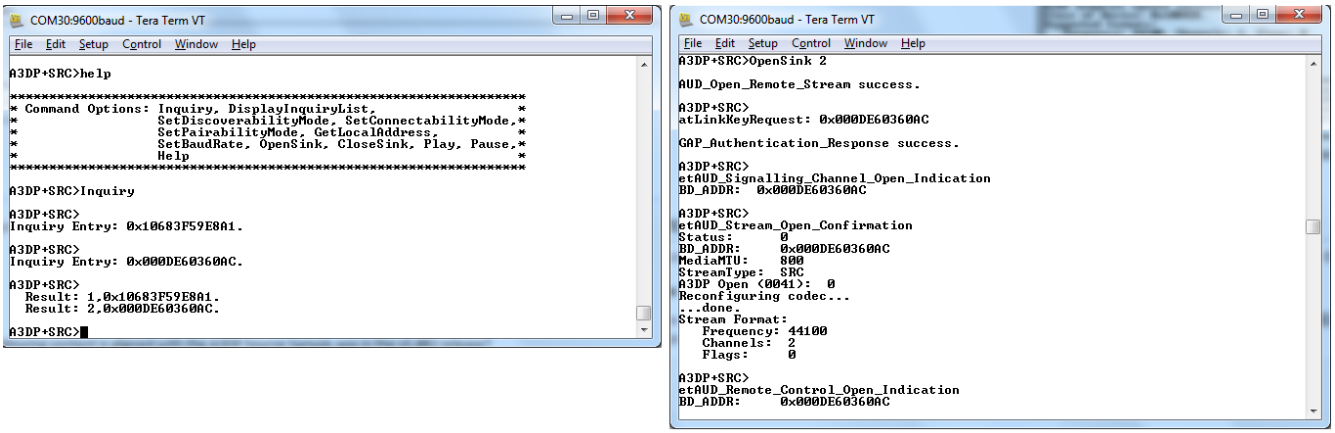

Figure 4-6. A3DP Source Inquiry and Open Command

3. Depending on the settings of the speaker device, it may initiate pairing as well.

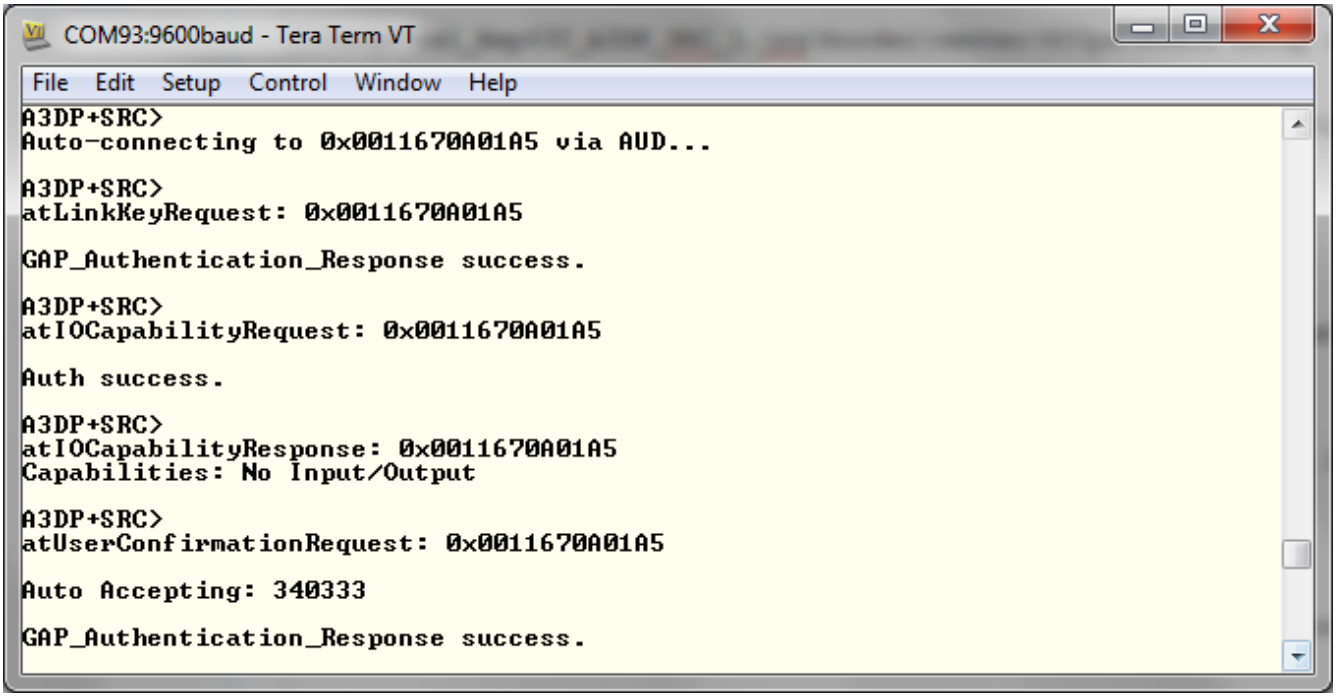

Figure 4-7. A3DP Source Terminal 1

4. The device will finish the connection and you should see a Stream State Change Confirmation and AUD Remote Control Open Indication events.
5. You can now transmit audio from the source device. Speak something through the mic and the speakers should play the output.

Connecting with the BT-MSP-AUDSNK board

Device 2 (Sink) setup

1. Make sure that the profile is flashed and the device is powered up and the speakers are connected.

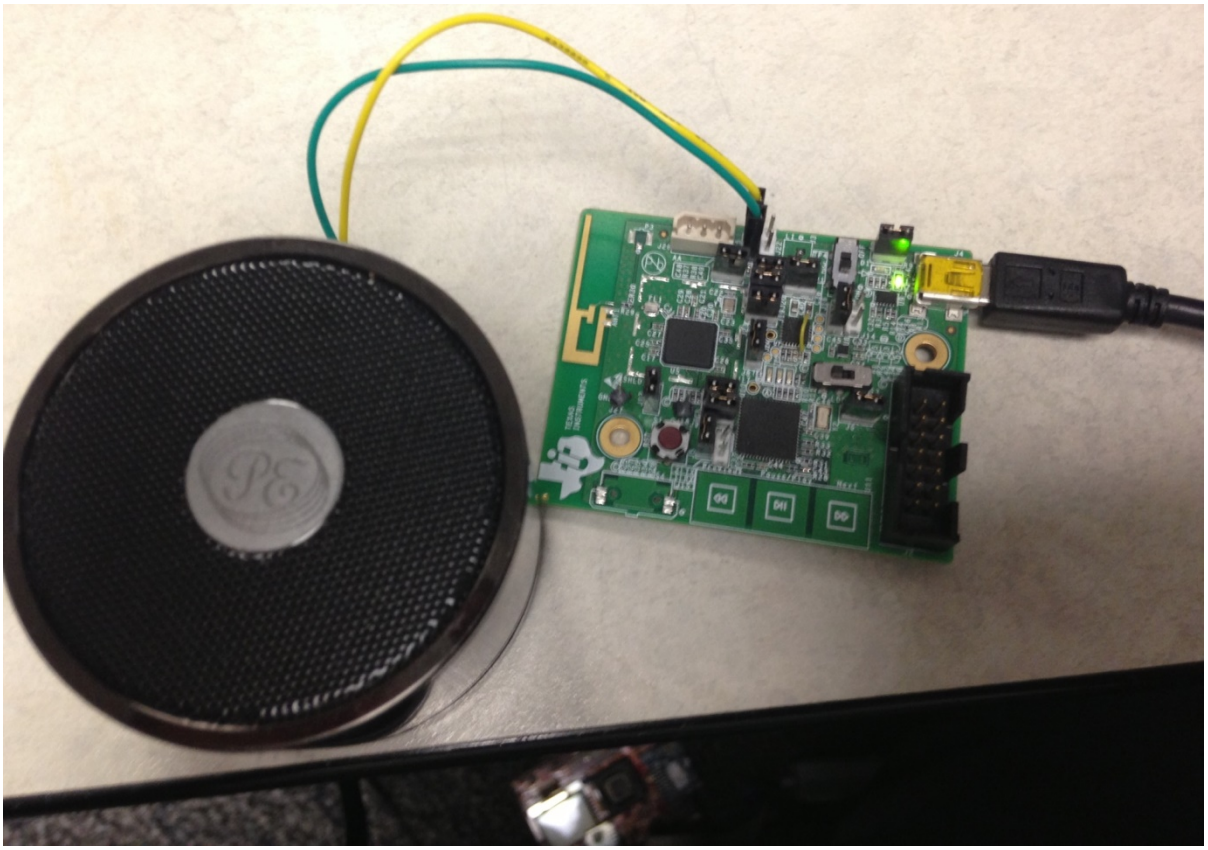

Figure 4-8. A3DP Source TI HW Device Setup

Device 1 (Source) Setup

1. Setup the device as shown in the previous section. The device should automatically perform an inquiry scan, pair and connect with the SNK board. The images below show the terminal (SNK on the left, SRC on the right) output of both of the devices.

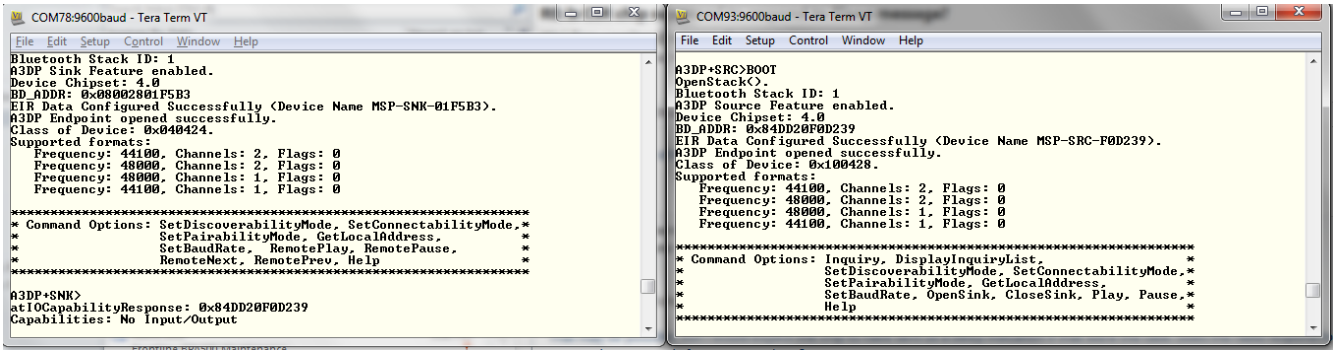

Figure 4-9. A3DP Source Initialization

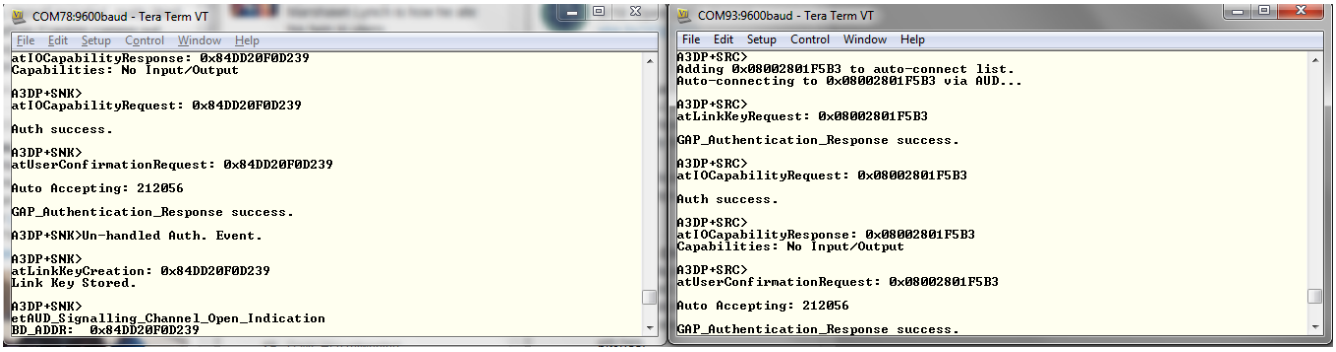

Figure 4-10. A3DP Source Device Pairing

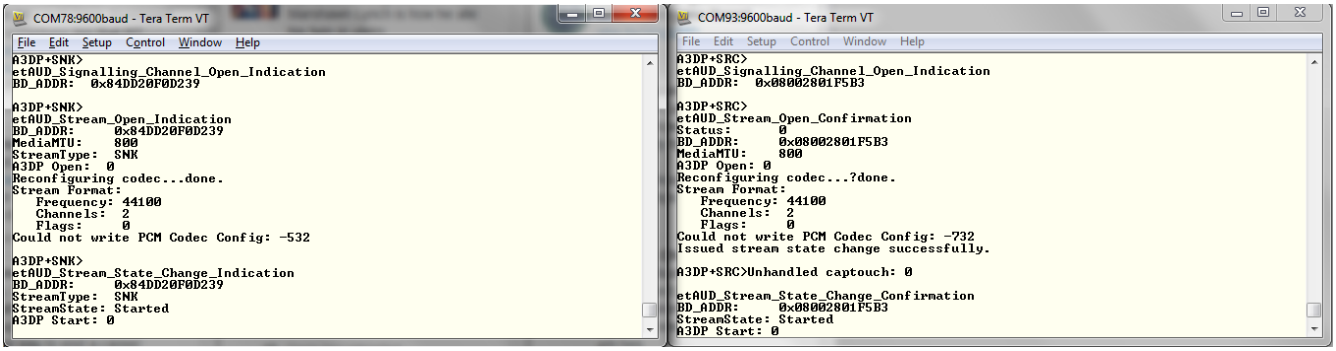

Figure 4-11. A3DP Source State Change

4.3 Application Commands

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands

A3DP Profile Commands

Description

The following command is responsible for checking changing the current baud rate used to talk to the Radio.

Note

This function ONLY configures the baud rate for a TI Bluetooth chipset.

Parameters

This command requires one parameter which is the Baud Rate that needs to be set.

Possible Return Values

(0) Success

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

API Call

VS_Update_UART_Baud_Rate(BluetoothStackID, (DWord_t)TempParam->Params[0].intParam);

API Prototype

int BTPSAPI VS_Update_UART_Baud_Rate(unsigned int BluetoothStackID, DWord_t BaudRate)

Description of API

The following function prototype represents the vendor specific function which is used to change the Bluetooth UART for the Local Bluetooth Device specified by the Bluetooth Protocol Stack that is specified by the Bluetooth Protocol Stack ID. The second parameter specifies the new baud rate to set. This change encompasses both changing the speed of the Bluetooth chip (by issuing the correct commands) and then, if successful, informing the HCI Driver of the change (so the driver can communicate with the Bluetooth device at the new baud rate). This function returns zero if successful or a negative return error code if there was an error.

OpenSink

Description

The following command is for opening a connection to a remote A2DP endpoint (Sink).

Parameters

This command requires one parameter which is the Inquiry Index of the Remote Bluetooth Device. This value can be found after an Inquiry or displayed when the command DisplayInquiryList is used. If the desired remote device does not appear in the list, it cannot be paired with.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2005) BTAUD_ERROR_ALREADY_CONNECTED

(-2008) BTAUD_ERROR_STREAM_NOT_INITIALIZED

(-2010) BTAUD_ERROR_STREAM_ALREADY_CONNECTED

(-2013) BTAUD_ERROR_STREAM_IS_ACTIVE

(-2032) BTAUD_ERROR_STREAM_CONNECTED

API Call

AUD_Open_Remote_Stream(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam - 1)], astSRC)

API Prototype

int BTPSAPI AUD_Open_Remote_Stream(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, AUD_Stream_Type_t StreamType)

Description of API

The following function is responsible for opening a remote streaming endpoint on the specified remote device. This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the requested Audio Manager is present, followed by the remote Bluetooth device AND the local Stream type. This function returns zero if successful or a negative return error code if there was an error.

CloseSink

Description

The following command is responsible for cleaning up AUD and the A3DP stream, if the stream is opened and/or playing.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the query.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2011) BTAUD_ERROR_STREAM_NOT_CONNECTED (-2014) BTAUD_ERROR_STREAM_IS_NOT_ACTIVE

API Call

AUD_Close_Stream(BluetoothStackID, RemoteSinkBD_ADDR, astSRC);

API Prototype

int BTPSAPI AUD_Close_Stream(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, AUD_Stream_Type_t StreamType)

Description of API

The following function is responsible for Closing a currently open stream endpoint on the local device. This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the requested Audio Manager is present, followed by the remote device address of the connected stream, followed by the stream endpoint type to close (local Stream Endpoint). This function returns zero if successful or a negative return error code if there was an error.

Play

Description

This command is responsible for handling a play command issued by the user.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the query.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2007) BTAUD_ERROR_UNABLE_TO_INITIALIZE_AVCTP

(-2008) BTAUD_ERROR_STREAM_NOT_INITIALIZED

(-2009) BTAUD_ERROR_UNABLE_TO_CONNECT_REMOTE_STREAM (-2011) BTAUD_ERROR_STREAM_NOT_CONNECTED

(-2014) BTAUD_ERROR_STREAM_IS_NOT_ACTIVE

(-2015) BTAUD_ERROR_STREAM_STATE_ALREADY_CURRENT

(-2016) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_STATE

(-2017) BTAUD_ERROR_STREAM_STATE_CHANGE_IN_PROGRESS

(-2018) BTAUD_ERROR_STREAM_FORMAT_CHANGE_IN_PROGRESS

(-2019) BTAUD_ERROR_UNSUPPORTED_FORMAT

(-2020) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_FORMAT

(-2021) BTAUD_ERROR_SAME_FORMAT

(-2022) BT_AUD_ERROR_RETRIEVING_SUPPORTED_FORMATS

(-2023) BTAUD_ERROR_UNABLE_TO_SEND_STREAM_DATA

(-2024) BTAUD_ERROR_UNABLE_TO_SEND_REMOTE_CONTROL_COMMAND

(-2026) BTAUD_ERROR_REMOTE_DEVICE_NOT_CONNECTED

(-2027) BTAUD_ERROR_REMOTE_CONTROL_NOT_CONNECTED

(-2028) BTAUD_ERROR_INVALID_REMOTE_CONTROL_DATA

(-2029) BTAUD_ERROR_REMOTE_CONTROL_ALREADY_CONNECTED

(-2030) BTAUD_ERROR_REMOTE_CONTROL_CONNECTION_IN_PROGRESS

(-2031) BTAUD_ERROR_REMOTE_CONTROL_NOT_INITIALIZED

API Call

AUD_Change_Stream_State(BluetoothStackID, RemoteSinkBD_ADDR, astSRC, astStreamStarted);

API Prototype

int BTPSAPI AUD_Change_Stream_State(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, AUD_Stream_Type_t StreamType, AUD_Stream_State_t StreamState)

Description of API

The following function is responsible for Changing the Stream State of a currently opened stream endpoint on the local device. This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the requested Audio Manager is present, followed by the remote device address of the connected stream, followed by the stream endpoint type to change the state of (local Stream Endpoint), followed by the new Stream Endpoint state. This function returns zero if successful or a negative return error code if there was an error.

Pause

Description

This command is responsible for handling a local pause command issued by the user.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the query.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2001) BTAUD_ERROR_NOT_INITIALIZED

(-2002) BTAUD_ERROR_INVALID_BLUETOOTH_STACK_ID

(-2007) BTAUD_ERROR_UNABLE_TO_INITIALIZE_AVCTP

(-2008) BTAUD_ERROR_STREAM_NOT_INITIALIZED

(-2009) BTAUD_ERROR_UNABLE_TO_CONNECT_REMOTE_STREAM

(-2011) BTAUD_ERROR_STREAM_NOT_CONNECTED

(-2014) BTAUD_ERROR_STREAM_IS_NOT_ACTIVE

(-2015) BTAUD_ERROR_STREAM_STATE_ALREADY_CURRENT

(-2016) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_STATE

(-2017) BTAUD_ERROR_STREAM_STATE_CHANGE_IN_PROGRESS

(-2018) BTAUD_ERROR_STREAM_FORMAT_CHANGE_IN_PROGRESS

(-2019) BTAUD_ERROR_UNSUPPORTED_FORMAT

(-2020) BTAUD_ERROR_UNABLE_TO_CHANGE_STREAM_FORMAT

(-2021) BTAUD_ERROR_SAME_FORMAT

(-2022) BT_AUD_ERROR_RETRIEVING_SUPPORTED_FORMATS

(-2023) BTAUD_ERROR_UNABLE_TO_SEND_STREAM_DATA

(-2024) BTAUD_ERROR_UNABLE_TO_SEND_REMOTE_CONTROL_COMMAND

(-2026) BTAUD_ERROR_REMOTE_DEVICE_NOT_CONNECTED

(-2027) BTAUD_ERROR_REMOTE_CONTROL_NOT_CONNECTED

(-2028) BTAUD_ERROR_INVALID_REMOTE_CONTROL_DATA

(-2029) BTAUD_ERROR_REMOTE_CONTROL_ALREADY_CONNECTED

(-2030) BTAUD_ERROR_REMOTE_CONTROL_CONNECTION_IN_PROGRESS

(-2031) BTAUD_ERROR_REMOTE_CONTROL_NOT_INITIALIZED

API Call

AUD_Change_Stream_State(BluetoothStackID, RemoteSinkBD_ADDR, astSRC, astStreamStarted);

API Prototype

int BTPSAPI AUD_Change_Stream_State(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, AUD_Stream_Type_t StreamType, AUD_Stream_State_t StreamState)

Description of API

The following function is responsible for Changing the Stream State of a currently opened stream endpoint on the local device. This function accepts as input the Bluetooth stack ID of the Bluetooth protocol stack that the requested Audio Manager is present, followed by the remote device address of the connected stream, followed by the stream endpoint type to change the state of (local Stream Endpoint), followed by the new Stream Endpoint state. This function returns zero if successful or a negative return error code if there was an error.

5 HFP Demo Guide

5.1 Demo Overview

The Hands free profile allows the user to demonstrate the use of Hands-free profile on an embedded device. The hands free profile is used to connect a headset or speakerphone with a mobile device to provide remote control and voice connections. The Hands-free profile supports two roles, Hands free and Audio Gateway. This document demonstrates how to use the hands free role of the profile.

Note

An external codec MUST be connected to the CC256x I2S/PCM interface to play and record audio. The same instructions can be used to run this demo on the MSP430 Platform.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as MSP-EXP430F5438 USB - Serial Port(COM x), Tiva Virtual COM Port (COM x) under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud (9600 for MSP430), 8, n, 1. Once connected, reset the device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

5.2 Demo Application

This section provides a description of how to use the demo application to connect two configured board and communicate over Bluetooth. Bluetooth HFP is a simple Client-Server connection process with one side operating in the Audio-Gateway role and the other operating in the Handsfree role. We will setup one of the boards as a Handsfree server and use an android phone as Client. Once connected, we can transmit data between the two devices over Bluetooth

Server setup on the demo application

1. Perform the steps mentioned earlier in Running the Bluetooth Code section to initialize the application.

```
*****************************************************************
* Command Options: Inquiry, DisplayInquiryList, Pair,           *
*                  EndPairing, PINCodeResponse, PassKeyResponse, *
*                  UserConfirmationResponse,                    *
*                  SetDiscoverabilityMode, SetConnectabilityMode,*
*                  SetPairabilityMode,                          *
*                  ChangeSimplePairingParameters,               *
*                  GetLocalAddress, GetLocalName, SetLocalName, *
*                  GetClassOfDevice, SetClassOfDevice,          *
*                  GetRemoteName, OpenHFServer, CloseHFServer   *
*                  ManageAudio, AnswerCall, HangUpCall, Close,  *
*                  Help                                         *
*****************************************************************
```

Figure 5-1. HFP Demo Start Terminal

2. Give a name for the MSP430 using the SetLocalName. In our example we give it a name of hfpserver.

```
HFRE16>SetLocalName hfpserver
Local Device Name set to: hfpserver.
```

Figure 5-2. HFP Demo Set Local Name

3. Open a HFPServer using the command, OpenHFServer <port number>. Here we use OpenHFServer 1 to open a port.

```
HFRE16>OpenHFServer 1
HFRE_Open_HandsFree_Server_Port: Function Successful.
HFRE_Register_HandsFree_SDP_Record: Function Successful.
```

Figure 5-3. HFP Demo Open Server

Client setup and device discovery

1. Open the bluetooth settings menu on the android phone Settings->Bluetooth . We should get a menu like this.
2. Hit on search for devices. The phone should begin looking for other bluetooth devices.

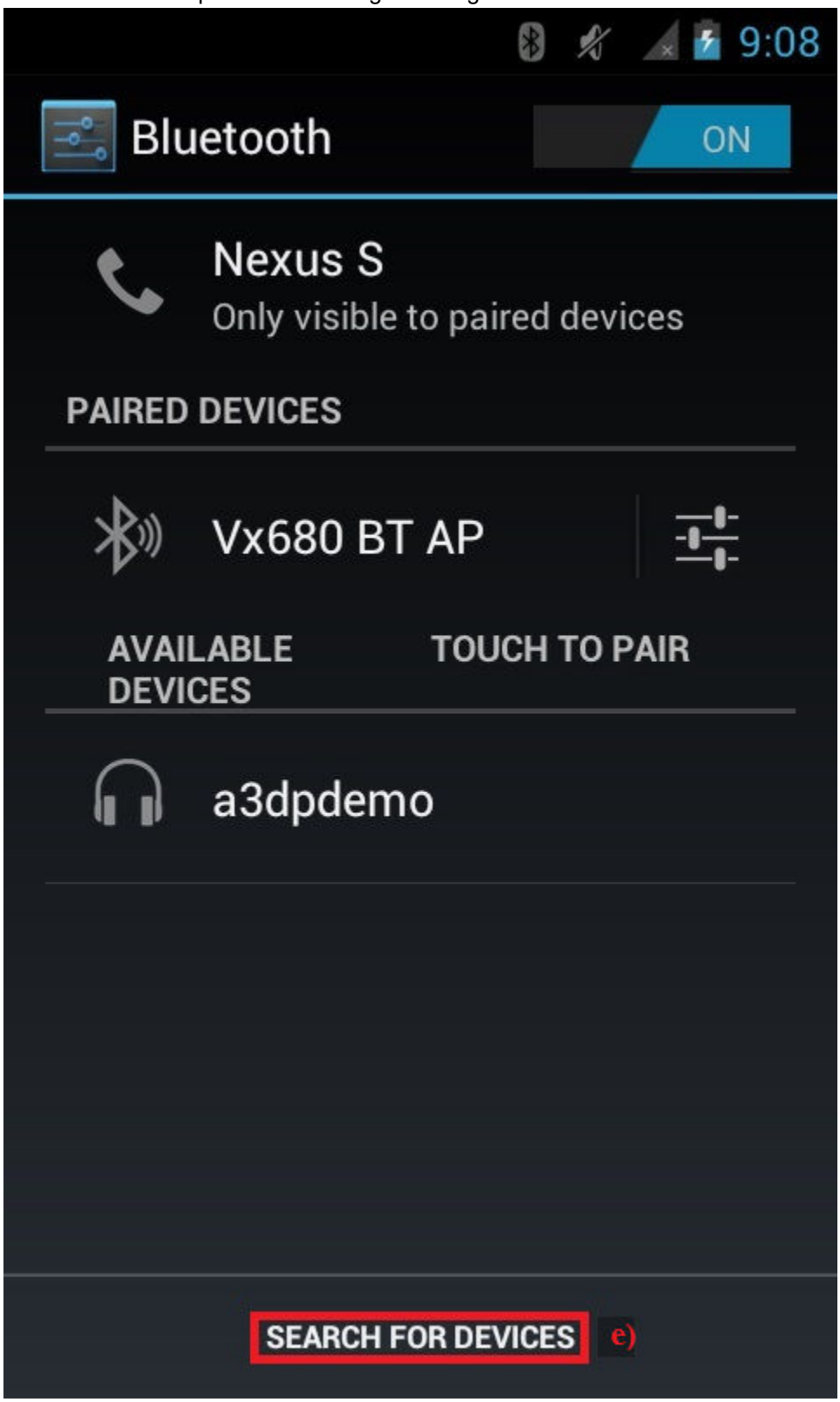

Figure 5-4. HFP Demo Discover Device

3. A Tiva HFPDemo Demo should appear like shown below in the picture. Click on the device to begin pairing.

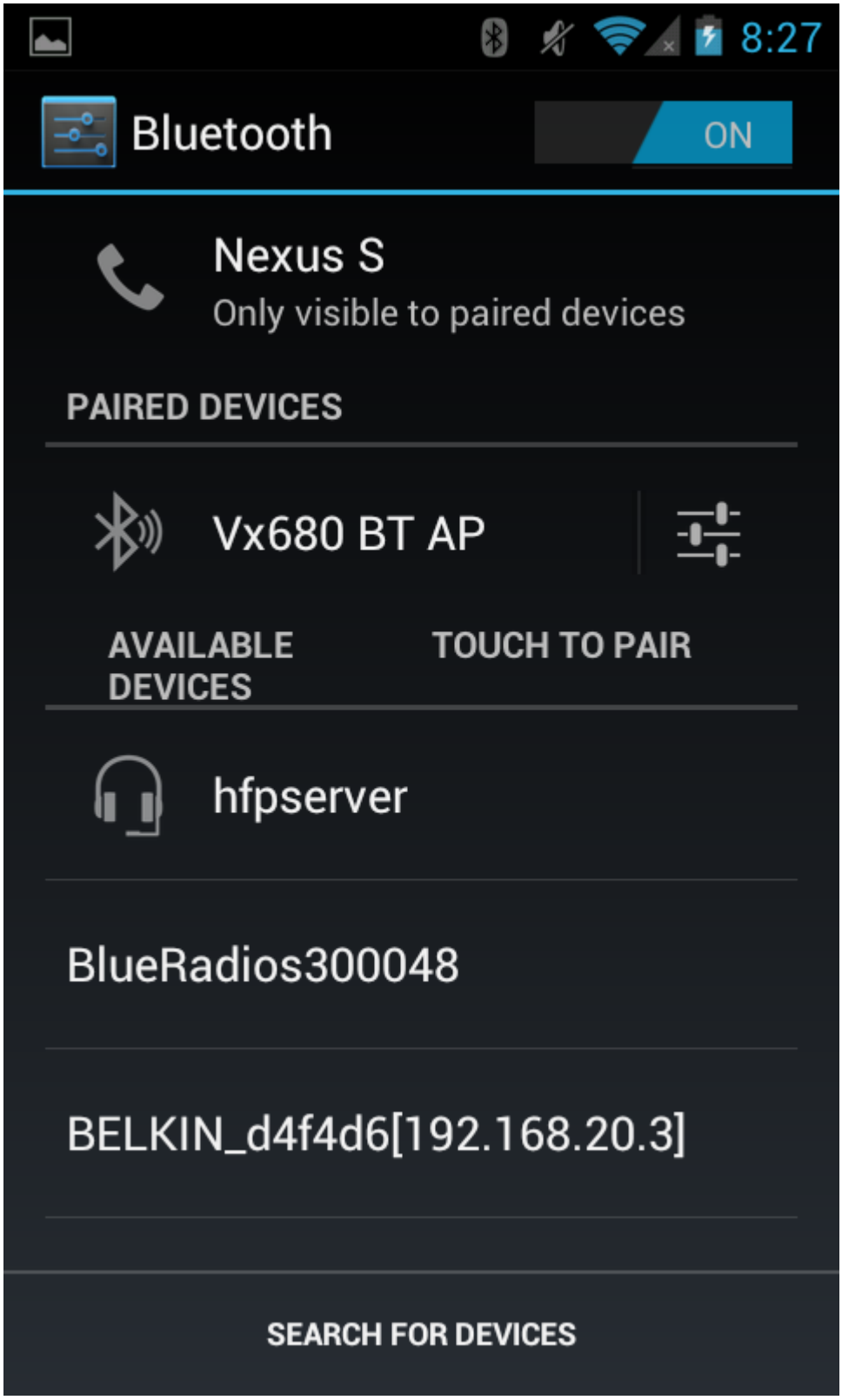

Figure 5-5. HFP Demo Pair

4. After the devices are paired, the device should show connected on the phone side and on the Tiva.

Figure 5-6. HFP Demo Connect

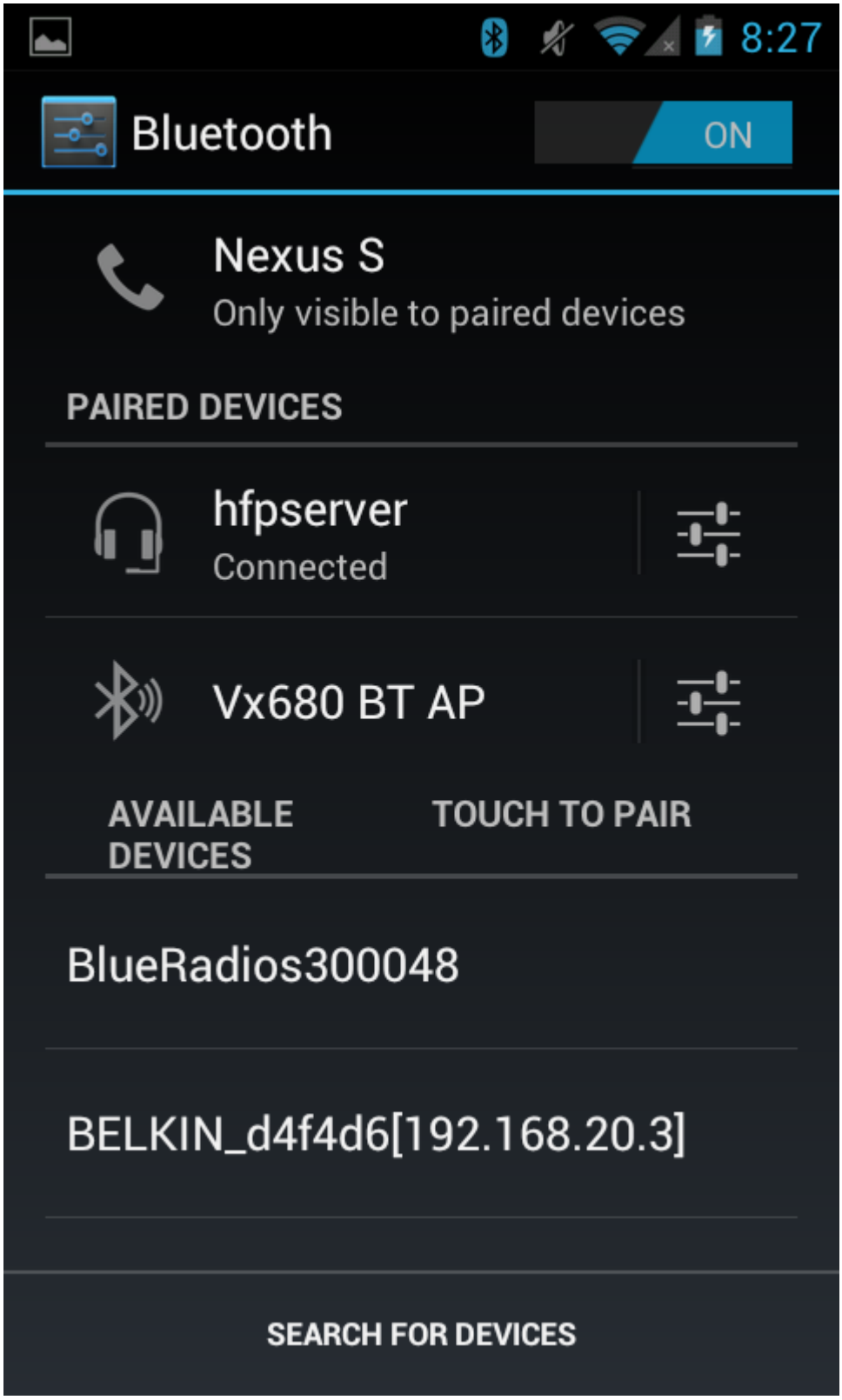

Figure 5-7. HFP Demo Status Indication Terminal

HandsFree Options

1. To Manage the audio connection to the headset. We use ManageAudio 0 to release the SCO connection and ManageAudio 1 to Setup the SCO connection.

```
HFRE16>
HFRE Open Port Indication, ID: 0x0001, Board: 0x00027232591C.

HFRE16>
HFRE Control Indicator Status Confirmation, ID: 0x0001, Description: SERVICE, Va
lue: FALSE.

HFRE16>
HFRE Control Indicator Status Confirmation, ID: 0x0001, Description: CALL, Value
: FALSE.

HFRE16>
HFRE Control Indicator Status Confirmation, ID: 0x0001, Description: CALL_SETUP,
 Value: 0.

HFRE16>
HFRE Control Indicator Status Confirmation, ID: 0x0001, Description: CALLSETUP,
Value: 0.

HFRE16>
HFRE Control Indicator Status Confirmation, ID: 0x0001, Description: CALLHELD, V
alue: FALSE.

HFRE16>
HFRE Control Indicator Status Confirmation, ID: 0x0001, Description: SIGNAL, Val
ue: 0.

HFRE16>
HFRE Control Indicator Status Confirmation, ID: 0x0001, Description: ROAM, Value
: FALSE.

HFRE16>
HFRE Open Service Level Connection Indication, ID: 0x0001
                    RemoteSupportedFeaturesValid: TRUE
                          RemoteSupportedFeatures: 0x00000511X
                  RemoteCallHoldMultipartySupport: 0x00000000X
HFRE_Enable Call Line Identification
```

Figure 5-8. HFP Demo Manage Audio Connection Terminal

```
HFRE16>ManageAudio 1
HFRE_Setup_Audio_Connection: Function Successful.

HFRE16>
HFRE Audio Connection Indication, ID: 0x0001, Status: 0x0000.
```

Figure 5-9. HFP Demo Manage Audio Connection Terminal 2

```
HFRE16>ManageAudio 0
HFRE_Release_Audio_Connection: Function Successful.
```

Figure 5-10. HFP Demo Answer Call Terminal

```
HFRE16>AnswerCall
HFRE_Answer_Incoming_Call: Function Successful.

HFRE16>
HFRE Command Result, ID: 0x0001, Type 0 Code 0.
```

Figure 5-11. HFP Demo Hang Up Terminal

2. To Close the HFserver use the CloseHFServer command.

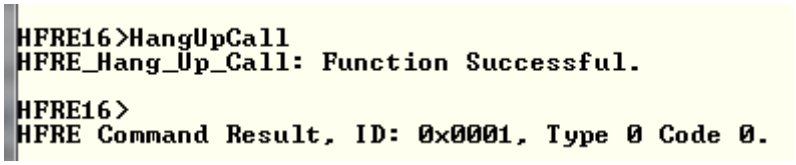

Figure 5-12. HFP Demo Close Server Terminal

5.3 Application Commands

TI's Bluetooth stack is implementation of the upper layers of the Bluetooth protocol stack. TI's Bluetooth stack provides a robust and flexible software development tool that implements the Bluetooth Protocols and Profiles above the Host Controller Interface (HCI). TI's Bluetooth stack's Application Programming Interface (API) provides access to the upper-layer protocols and profiles and can interface directly with the Bluetooth chips.

This page describes the various commands that a user of the application can use. Each command is a wrapper over a TI's Bluetooth stack API which gets invoked with the parameters selected by the user. This is a subset of the APIs available to the user. TI's Bluetooth stack API documentation (TI_Bluetooth_Stack_Version-Number\Documentation) describes all of the API's in detail.

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands

OpenHFServer

Description

The following function is responsible for opening a Serial Port Server on the Local Device. This function opens the Serial Port Server on the specified RFCOMM Channel. This function returns zero if successful, or a negative return value if an error occurred.

Parameters

The command only requires one parameter. This parameter is the Port Number.

Command Call Examples

"OpenHFServer 4" Attempts to Open a Handsfree Server at Port Number #4.

"OpenHFServer 1" Attempts to Open a Handsfree Server at Port Number #1.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Open_HandsFree_Server_Port(BluetoothStackID, TempParam->Params[0].intParam, HFRE_SUPPORTED_FEATURES, 0, NULL, HFRE_Event_Callback, (unsigned long)0)

API Prototype

int BTPSAPI HFRE_Open_HandsFree_Server_Port(unsigned int BluetoothStackID, unsigned int ServerPort, unsigned long SupportedFeaturesMask, unsigned int NumberAdditionalIndicators, char *AdditionalSupportedIndicators[], HFRE_Event_Callback_t EventCallback, unsigned long CallbackParameter)

Description of API

The following function is responsible for Opening a Hands-Free Server on the specified Bluetooth SPP Serial Port. This function accepts as input the Bluetooth Stack ID of the Bluetooth Stack Instance to use for the Hands-Free Server, the Local Serial Port Server Number to use, a bit mask that specifies the features in which the Hands-Free unit supports, the Number of Additional Indicators, a list of Additional Indicators to Support, and the HFRE Event Callback function (and parameter) to associate with the specified Hands-Free Port. The ServerPort parameter *MUST* be between SPP_PORT_NUMBER_MINIMUM and SPP_PORT_NUMBER_MAXIMUM. This function returns a positive, non-zero, value if successful or a negative return error code if an error occurs. A successful return code will be a HFRE Port ID that can be used to reference the Opened HFRE Port in ALL other functions in this module except for the HFRE_Register_Audio_Gateway_SDP_Record() function which is specific to an Audio Gateway Server NOT a Hands-Free Server. Once a Server HFRE Port is opened, it can only be Un-Registered via a call to the HFRE_Close_Server_Port() function (passing the return value from this function). The HFRE_Close_Port() function can be used to Disconnect a Client from the Server Port (if one is connected, it will NOT Un-Register the Server Port however).

Note

The Mandatory Hands-Free Indicators (call, service, and call_setup) are automatically added to the list and need not be specified as additional indicators.

CloseHFServer

Description

The following function is responsible for closing a Serial Port Server that was previously opened via a successful call to the OpenServer() function. This function returns zero if successful or a negative return error code if there was an error.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of CloseHFServer.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Close_Server_Port(BluetoothStackID, HFServerPortID)

API Prototype

int BTPSAPI HFRE_Close_Server_Port(unsigned int BluetoothStackID, unsigned int HFREPortID)

Description of API

The following function is responsible for Un-Registering a HFRE Port Server (which was Registered by a successful call to either the HFRE_Open_HandsFree_Server_Port() or the HFRE_Open_Audio_Gateway_Server_Port() function). This functionaccepts as input the Bluetooth Stack ID of the Bluetooth Protocol Stack that the HFRE Port specified by the Second Parameter is valid for. This function returns zero if successful, or a negative return error code if an error occurred (see BTERRORS.H). Note that this function does NOT delete any SDP Service Record Handles.

ManageAudio

Description

The following function is responsible for setting up or releasing an audio connection. This function returns zero on successful execution and a negative value on all errors.

Parameters

The Manage Audio command requires only one parameter to which is is an integer value that represents the ManageAudio mode. This value must be specified as 0 (for Release) or 1 (for Setup)

Command Call Examples

"ManageAudio 0" Attempts to Release the Audio Connection.

"ManageAudio 1" Attempts to Setup the Audio Connection.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Setup_Audio_Connection(BluetoothStackID, HFServerPortID)

or HFRE_Release_Audio_Connection(BluetoothStackID, HFServerPortID)

API Prototype

int BTPSAPI HFRE_Setup_Audio_Connection(unsigned int BluetoothStackID, unsigned int HFREPortID)

or int BTPSAPI HFRE_Release_Audio_Connection(unsigned int BluetoothStackID, unsigned int HFREPortID)

Description of API

This function is responsible for Setting Up an Audio Connection between the Local and Remote Device. This function may be used by either an Audio Gateway or a Hands-Free unit for which a valid Service Level Connection Exists. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. This function returns zero if successful or a negative return error code if there was an error.

(or) This function is responsible for Releasing an Audio Connection which was previously established by the Remote Device or by a call to the HFRE_Setup_Audio_Connection() function. This function may be used by either an Audio Gateway or a Hands-Free unit. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. This function returns zero if successful or a negative return error code if there was an error.

AnswerCall

Description

The following function is responsible for sending the command to Anwser an Incoming Call on the Remote Audio Gateway. This function returns zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of AnswerCall .

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Answer_Incoming_Call(BluetoothStackID, HFServerPortID)

API Prototype

int BTPSAPI HFRE_Answer_Incoming_Call(unsigned int BluetoothStackID, unsigned int HFREPortID)

Description of API

This function is responsible for sending the command to Answer an Incoming call on a Remote Audio Gateway. This function may only be performed by Hands-Free units for which a valid Service Level Connection exists. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. This function return zero if successful or a negative return error code if there was an error.

HangupCall

Description

The following function is responsible for sending the command to HangUp ongoing calls or reject incoming calls on the Remote Audio Gateway. This function returns zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of HangupCall

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Hang_Up_Call(BluetoothStackID, HFServerPortID)

API Prototype

int BTPSAPI HFRE_Hang_Up_Call(unsigned int BluetoothStackID, unsigned int HFREPortID)

Description of API

This function is responsible for sending a Hang-Up Command to the Remote Audio Gateway. This function may be used to Reject an incoming call, or to terminate an ongoing call. This function may only be performed by Hands-Free units for which a valid Service Level Connection exists. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. This function returns zero if successful or a negative return error code if there was an error.

Close

Description

The following function is responsible for closing any open HFP ports. This function returns zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of Close.

Possible Return Values

(0) A3DP Endpoint opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Close_Port(BluetoothStackID, HFServerPortID)

API Prototype

int BTPSAPI HFRE_Close_Port(unsigned int BluetoothStackID, unsigned int HFREPortID)

Description of API

The following function exists to close a HFRE Port that was previously opened by any of the following mechanisms:

- Successful call to HFRE_Open_Remote_HandsFree_Port() function.

- Successful call to HFRE_Open_Remote_Audio_Gateway_Port() function.

- Incoming open request (Hands-Free or Audio Gateway) which the server was opened with either the HFRE_Open_HandsFree_Server_Port() or the HFRE_Open_Audio_Gateway_Server_Port() functions.

This function accepts as input the Bluetooth Stack ID of the Bluetooth Stack which the Open HFRE Port resides and the HFRE Port ID (return value from one of the above mentioned Open functions) of the Port to Close. This function returns zero if successful, or a negative return value if there was an error. This function does NOT Un-Register a HFRE Server Port from the system, it ONLY disconnects any connection that is currently active on the Server Port. The HFRE_Close_Server_Port()function can be used to Un-Register the HFRE Server Port.

6 HFP Audio Gateway Demo Guide

6.1 Demo Overview

The Hands-free profile allows the user to demonstrate the use of Hands-free profile on embedded device. The Hands-free profile is used to connect a headset or speakerphone with a mobile device to provide remote control and voice connections. The Hands-free profile supports two roles, Hands-free and Audio Gateway. This document demonstrates how to use the Audio Gateway role of the profile.

Note

The same instructions can be used to run this demo on the MSP432 or STM32F4 Platform. An external codec MUST be connected to the CC256x I2S/PCM interface to play and record audio.

It is recommended that the user visits the kit setup Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as, XDS110 Class Application/User UART (COM x) for MSP432 under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud, 8, n, 1. Once connected, reset the device using Reset S3 button and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands. This device will become the Audio Gateway.

6.2 Demo Application

This section provides a description of how to use the demo application to connect two configured board and communicate over Bluetooth. The Bluetooth HFP is a simple Client-Server connection process with one side operating in the Audio-Gateway role and the other operating in the Hands-free role. We will setup one of the boards as a Audio-Gateway client (HFPAG Demo acts as Audio gateway) and use an another board as Hands-free server (using HFP Demo). we Will see how to initiate connection and send indications between the two devices over Bluetooth.

Now use the second board and follow the steps as per either HFP Demo when using Tiva or MSP430 as the server or HFPDemo_HF for MSP432 or STM32F4 as the server. Perform the steps prior to running the Audio-Gateway Bluetooth code on the first board. The second device that is connected to the computer will be the Hands-free Server.

Server setup using HFP demo application

1. Perform the steps mentioned earlier in Running the Bluetooth Code section to initialize the application.
2. Give a name for the platform by issuing the SetLocalName. In our example we give it a name of hfpserver.
3. Open a HFPServer by issuing the command, OpenHFServer. Here we use OpenHFServer to open port 1, the default first port.

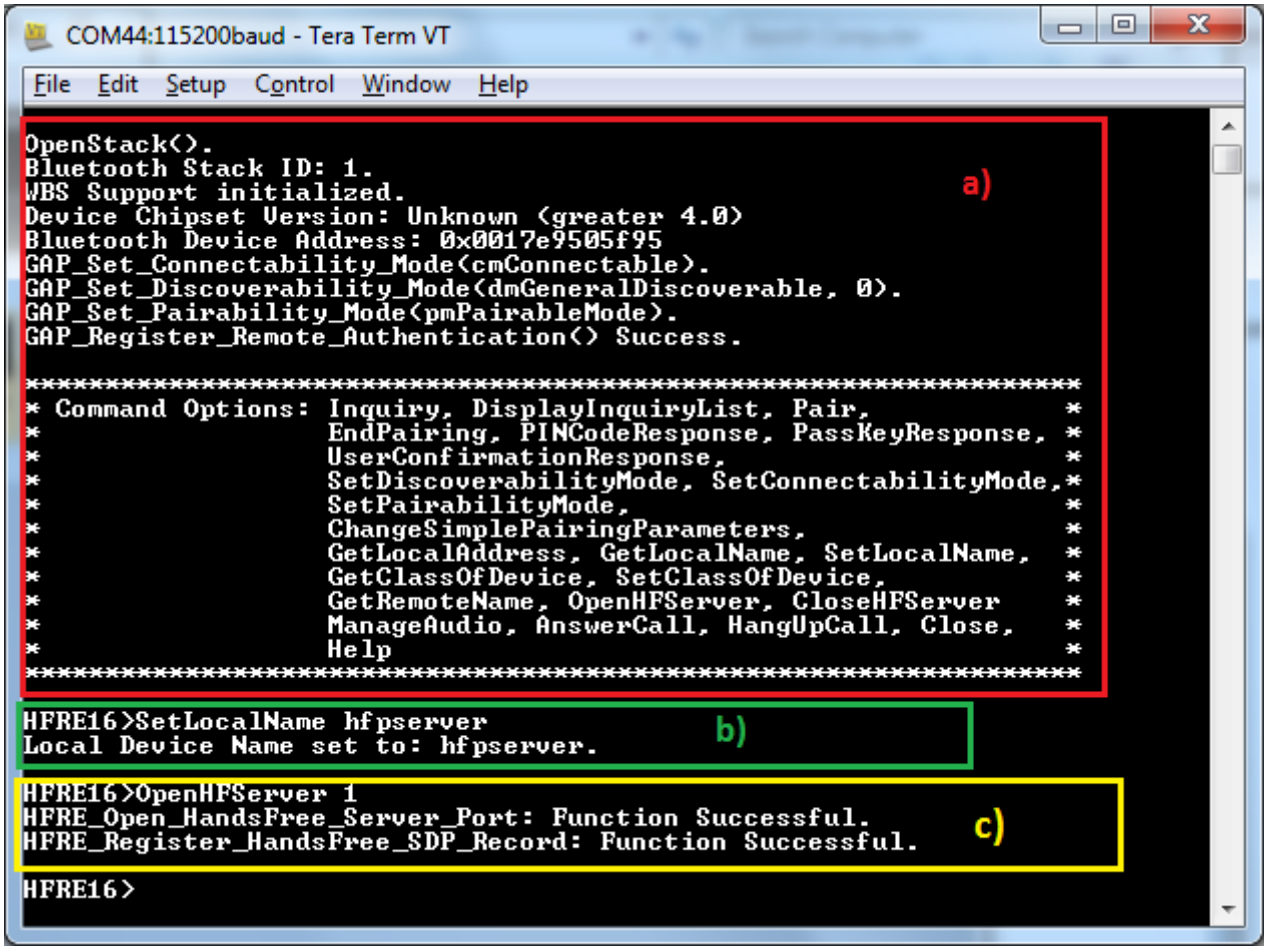

Figure 6-1. HFP AG Demo Start Server

Client setup on the demo application

1. Perform the steps mentioned earlier in "Running the Bluetooth Code" section to initialize the application.

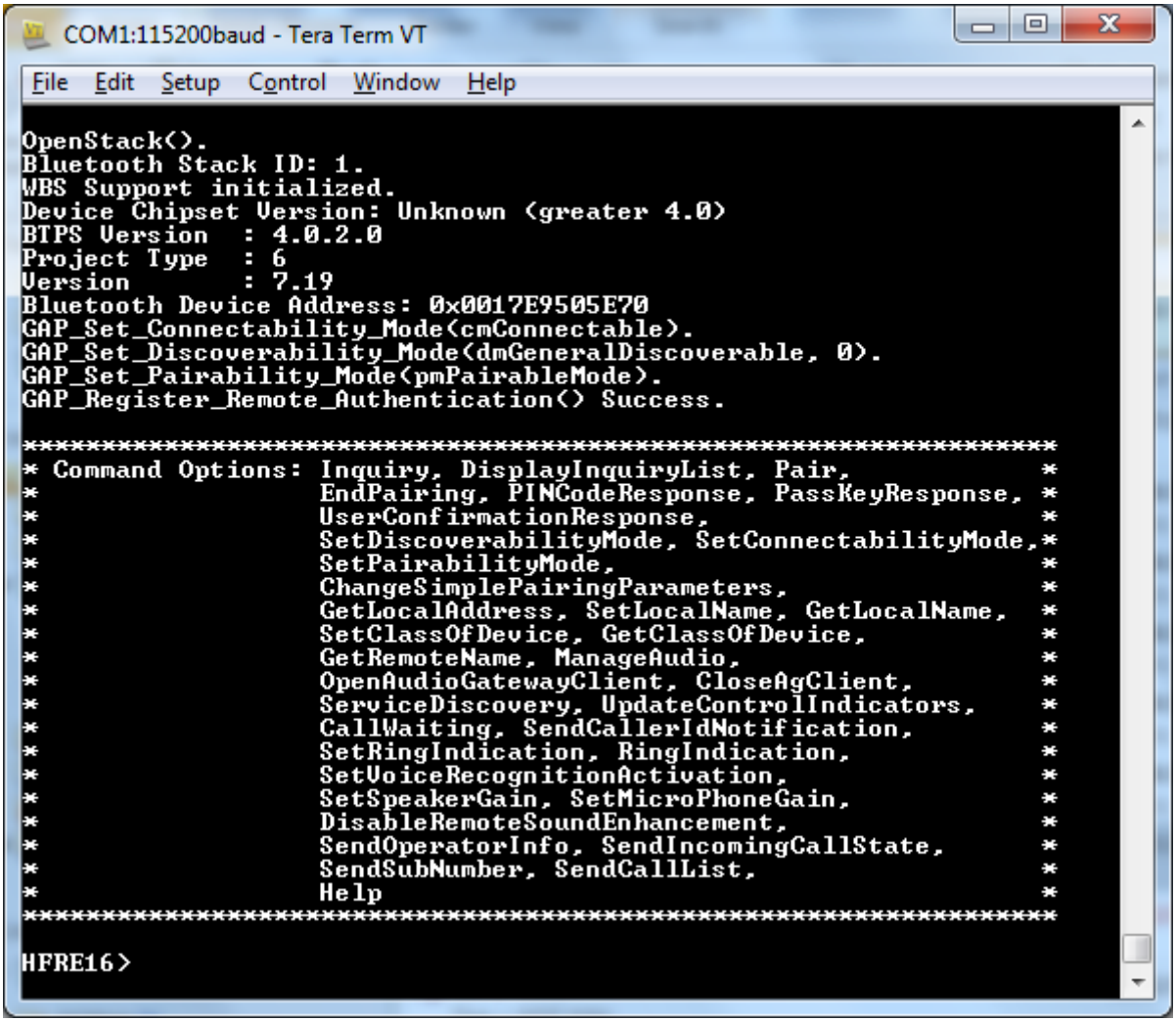

Figure 6-2. HFP AG Demo Start Client

2. Issue the Inquiry command for the HFP server.

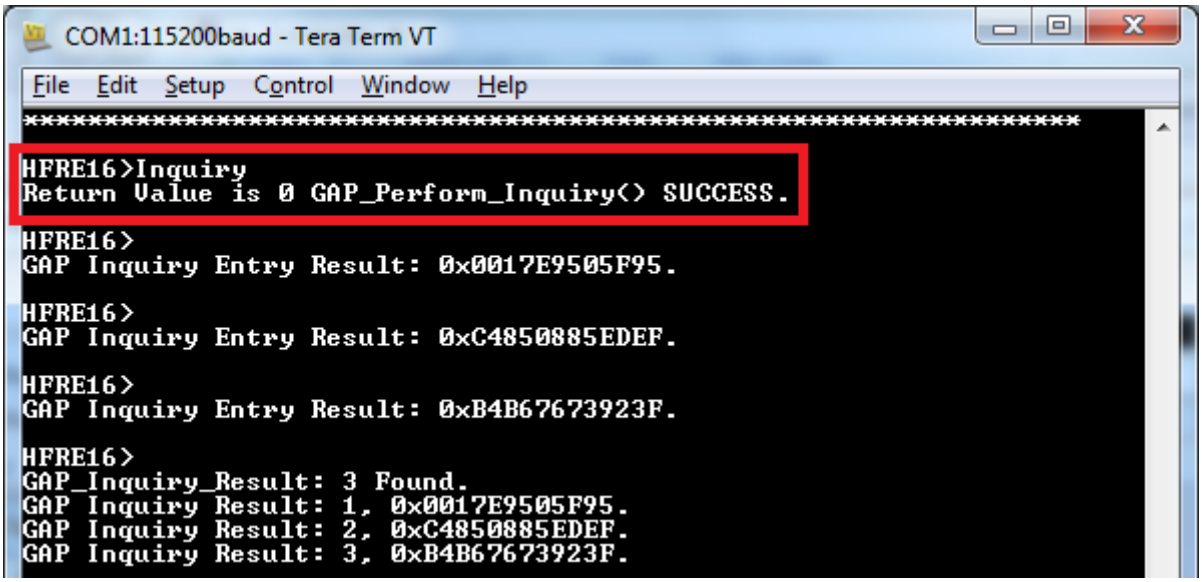

Figure 6-3. HFP AG Demo Inquiry

3. Discover servers of the remote HFP server by issuing the ServiceDiscovery 1 12 , command to get the port number.

Note

The port ID on the remote Hands free device is 0x01 (The Unsigned int), as highlighted in the image below from the Attribute ID 0x0004. This port ID is used in the following OpenAudioGatewayClient command as its second parameter.

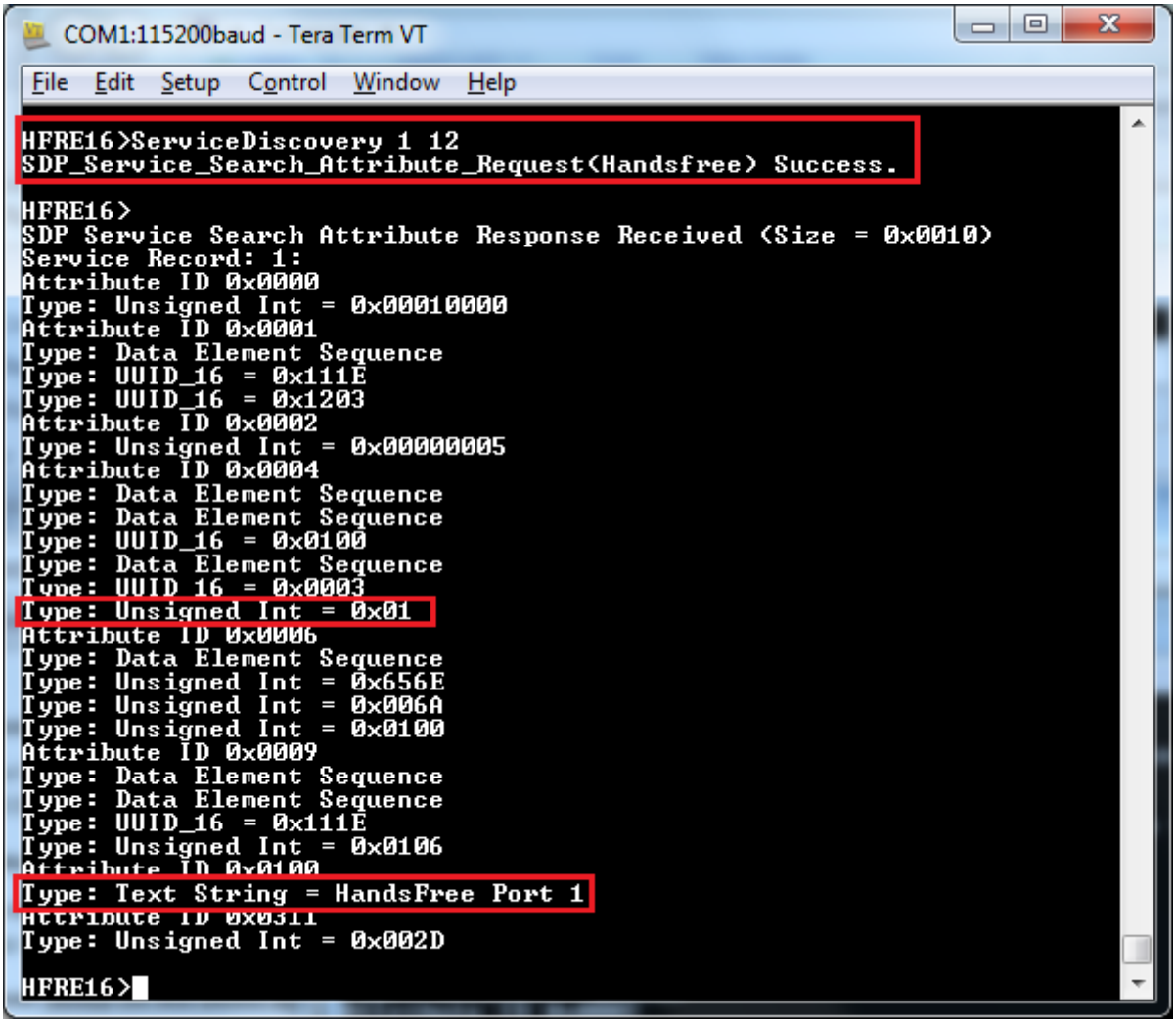

Figure 6-4. HFP AG Demo Discovery

4. Initiate connection to the remote HFP server by isssuing the OpenAudioGatewayClient 1 1 command.

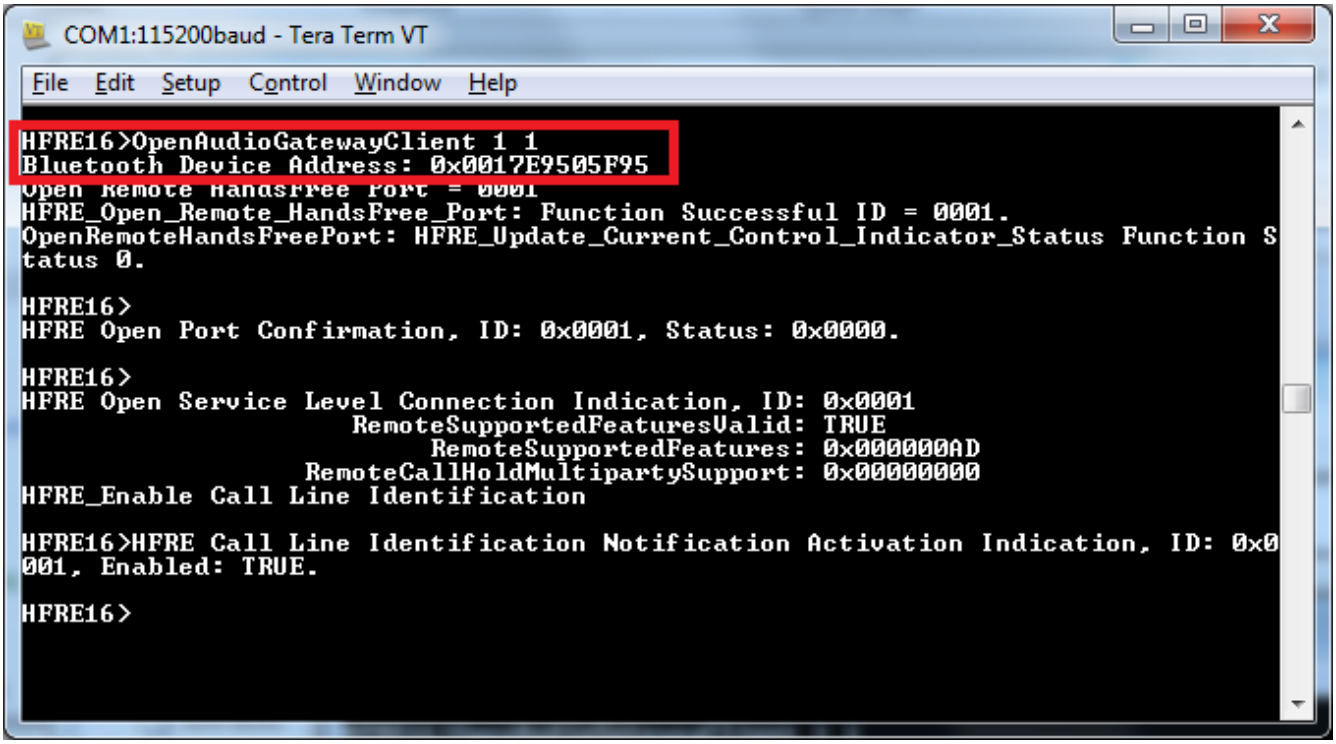

Figure 6-5. HFP AG Demo Open Audio Gateway

You will see the output below from the HFP server.

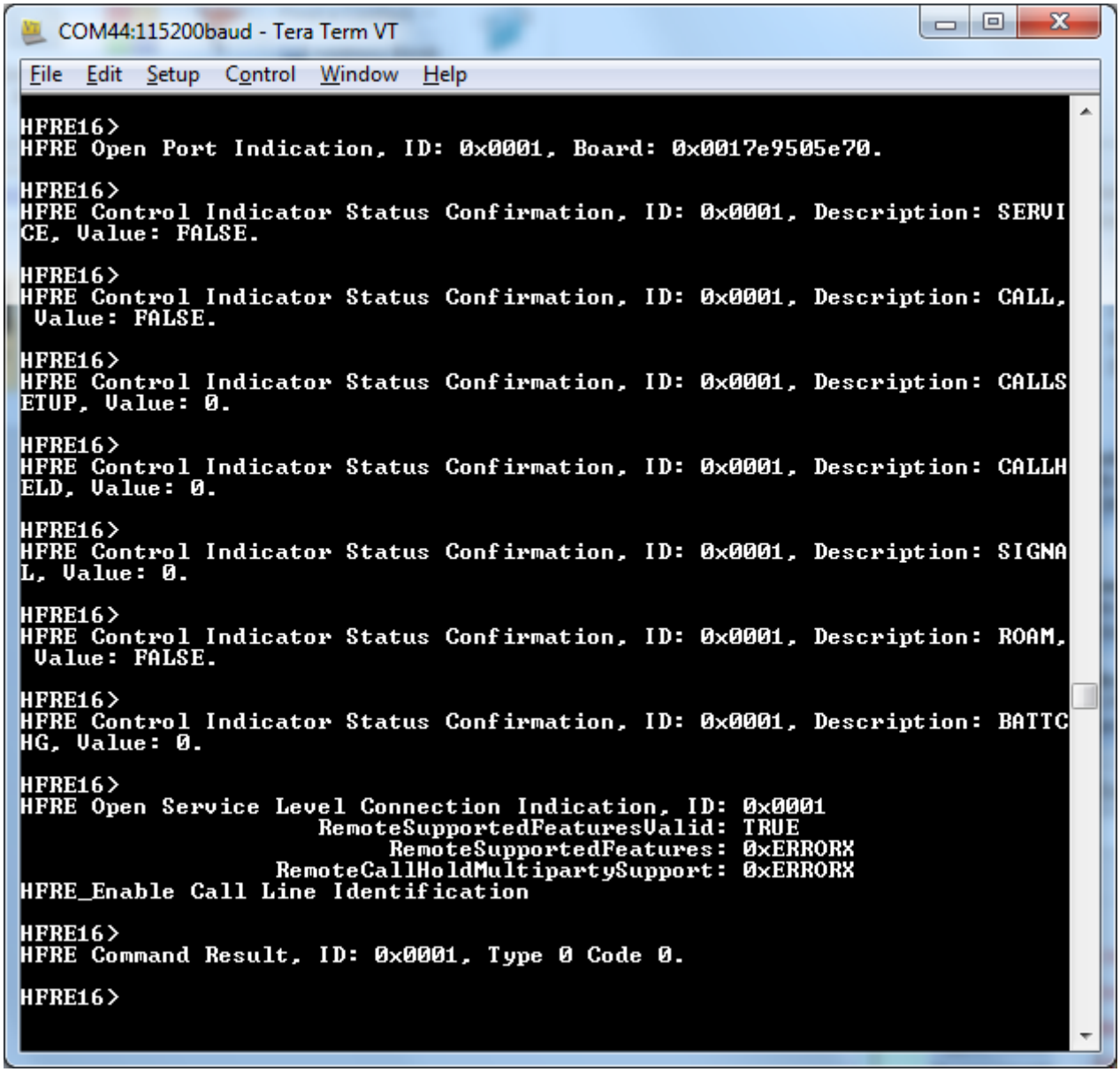

Figure 6-6. HFP AG Status Indication

5. Sending indications: Issue the RingIndication or SendCallerIdNotification +9198787899889 commands.

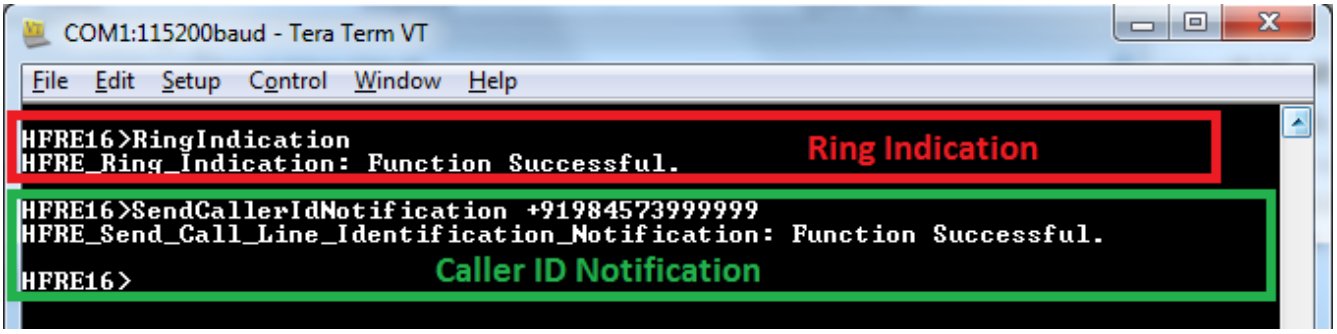

Figure 6-7. HFP AG Demo Ring Indication

You will see the below output from the HFP server.

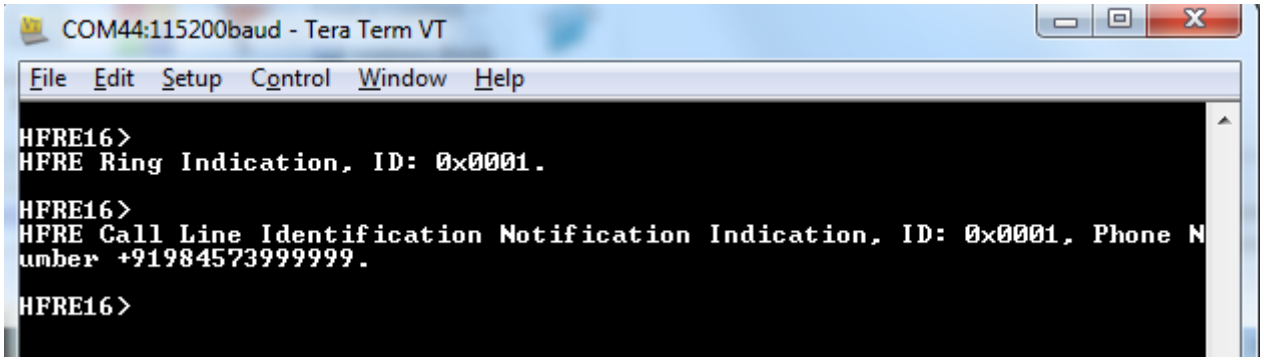

Figure 6-8. HFP AG Demo Ring Indication Server

6. Initiate audio connection/Disconnection and closing the HFP connection by issuing the: ManageAudio <STATE> and CloseAgClient commands.

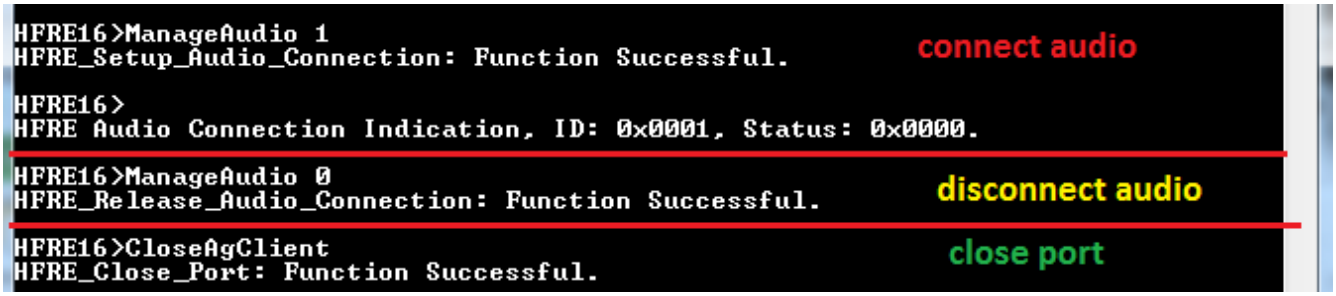

Figure 6-9. HFP AG Demo Manage Audio Cycle

You will see the below output from the HFP server.

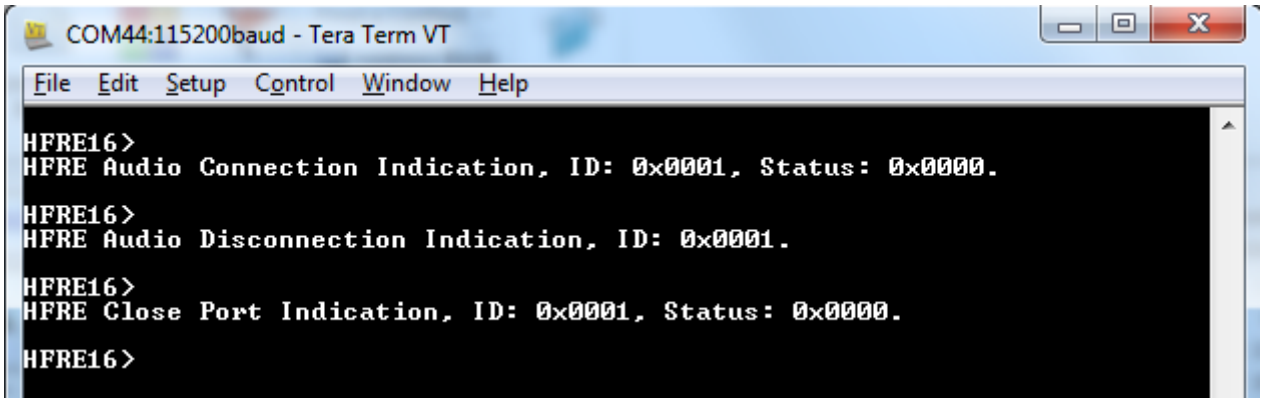

Figure 6-10. HFP AG Demo Server Indications

Example: Audio gateway with a commercial headset

This demonstrates setting up the client to connect to a commercial headset.

1. Perform the steps mentioned earlier in "Running the Bluetooth Code" section to initialize the application.

   OpenStack().

   Bluetooth Stack ID: 1

   WBS support enabled.

   Device Chipset: Unknown (greater 4.1)

   BTPS Version : 4.2.1.0

   Project Type : 6

   FW Version : 12.12

App Name : HFPDemo_AG

App Version : 0.3

LOCAL BD_ADDR: 0x88C255D1D645

GAP_Set_Connectability_Mode(cmConnectable).

GAP_Set_Discoverability_Mode(dmGeneralDiscoverable, 0).

GAP_Set_Pairability_Mode(pmPairableMode).

GAP_Register_Remote_Authentication() Success.

```
**********************************************************************
* Command Options: Inquiry, DisplayInquiryList, Pair, *
* EndPairing, PINCodeResponse, PassKeyResponse, *
* UserConfirmationResponse, *
* SetDiscoverabilityMode, SetConnectabilityMode,*
* SetPairabilityMode, *
* ChangeSimplePairingParameters, *
* GetLocalAddress, SetLocalName, GetLocalName, *
* SetClassOfDevice, GetClassOfDevice, *
* GetRemoteName, ManageAudio, *
* OpenAudioGatewayClient, CloseAgClient, *
* ServiceDiscovery, UpdateControlIndicators, *
* CallWaiting, SendCallerIdNotification, *
* SetRingIndication, RingIndication, *
* SetVoiceRecognitionActivation, *
* SetSpeakerGain, SetMicroPhoneGain, *
* DisableRemoteSoundEnhancement, *
* SendOperatorInfo, SendIncomingCallState, *
* SendSubNumber, SendCallList, *
* Help *
**********************************************************************
```

2. Issue the Inquiry command for the HFP server.

```
HFP AG>Inquiry
Return Value is 0 GAP_Perform_Inquiry() SUCCESS.
HFP AG>
GAP Inquiry Entry Result: 0x244B03F712D3.
HFP AG>
GAP Inquiry Entry Result: 0x3402862CCAE9.
HFP AG>
GAP Inquiry Entry Result: 0x340286605044.
HFP AG>
```

GAP Inquiry Entry Result: 0x484520902A4E.

HFP AG>

GAP Inquiry Entry Result: 0x08DF1F99F8D0.

HFP AG>

GAP Inquiry Entry Result: 0x002500F84FAB.

HFP AG>

GAP_Inquiry_Result: 6 Found.

GAP Inquiry Result: 1, 0x244B03F712D3.

GAP Inquiry Result: 2, 0x3402862CCAE9.

GAP Inquiry Result: 3, 0x340286605044.

GAP Inquiry Result: 4, 0x484520902A4E.

GAP Inquiry Result: 5, 0x08DF1F99F8D0.

GAP Inquiry Result: 6, 0x002500F84FAB.

3. Discover services of the remote HFP server by issuing the ServiceDiscovery 5 12 , command to get the port number.

Note

The port ID on the remote Hands free device is 0x0A (The Unsigned int), from the Attribute ID 0x0004. This port ID is used in the following OpenAudioGatewayClient command as its second parameter after being converted to its decimal equivilant (10).

HFP AG>ServiceDiscovery

Usage: SERVICEDISCOVERY [Inquiry Index] [Profile Index] [16/32 bit UUID (Manual only)].

Profile Index:

a. Manual (MUST specify 16/32 bit UUID)
b. L2CAP
c. Advanced Audio
d. A/V Remote Control
e. Basic Imaging
f. Basic Printing
g. Dial-up Networking
h. FAX
i. File Transfer
j. Hard Copy Cable Repl.
k. Health Device
l. Headset
m. Handsfree
n. HID
o. LAN Access
p. Message Access
q. Object Push
r. Personal Area Network
s. Phonebook Access
t. SIM Access
u. Serial Port
v. IrSYNC

Function Error.

HFP AG>ServiceDiscovery 5 12

SDP_Service_Search_Attribute_Request(Handsfree) Success.

HFP AG>

SDP Service Search Attribute Response Received (Size = 0x0010)

Service Record: 1:

Attribute ID 0x0000

Type: Unsigned Int = 0x00010000

Attribute ID 0x0001

Type: Data Element Sequence

Type: UUID_16 = 0x111E

Type: UUID_16 = 0x1203

Attribute ID 0x0004

Type: Data Element Sequence

Type: Data Element Sequence

Type: UUID_16 = 0x0100

Type: Data Element Sequence

Type: UUID_16 = 0x0003

Type: Unsigned Int = 0x0A

Attribute ID 0x0006

Type: Data Element Sequence

Type: Unsigned Int = 0x656E

Type: Unsigned Int = 0x006A

Type: Unsigned Int = 0x0100

Attribute ID 0x0009

Type: Data Element Sequence

Type: Data Element Sequence

Type: UUID_16 = 0x111E

Type: Unsigned Int = 0x0106

Attribute ID 0x0100

Type: Text String = Hands-Free unit

Attribute ID 0x0311

Type: Unsigned Int = 0x003F

4. Initiate connection to the remote HFP server by isssuing the OpenAudioGatewayClient 5 10 command.

HFP AG>OpenAudioGatewayClient

Usage: OPENAUDIOGATEWAYCLIENT [Inquiry Index] [Port Number].

Function Error.

HFP AG>OpenAudioGatewayClient 5 10

Bluetooth Device Address: 0x08DF1F99F8D0

Open Remote HandsFree Port = 000A

HFRE_Open_Remote_HandsFree_Port: Function Successful ID = 0001.

OpenRemoteHandsFreePort: HFRE_Update_Current_Control_Indicator_Status Function Status 0.

You will see the output below from the HFP server

HFP AG>

atLinkKeyRequest: 0x08DF1F99F8D0

GAP_Authentication_Response() Success.

HFP AG>

atPINCodeRequest: 0x08DF1F99F8D0

Respond with the command: PINCodeResponse

HFP AG>PINCodeResponse 0000

PINCodeResponse.

GAP_Authentication_Response(), Pin Code Response Success.

HFP AG>

atLinkKeyCreation: 0x08DF1F99F8D0

Link Key: 0x4AF49AD7072771919BAC62840F1F985D

Link Key Stored locally.

HFP AG>

HFRE Open Port Confirmation, ID: 0x0001, Status: 0x0000.

HFP AG>

HFRE Available Codec List Indiocation, ID: 0x0001 NumCodecs: 2 [1 2]

HFP AG>

HFRE Open Service Level Connection Indication, ID: 0x0001

RemoteSupportedFeaturesValid: TRUE

RemoteSupportedFeatures: 0x000000BF

RemoteCallHoldMultipartySupport: 0x00000000

HFRE_Enable Call Line Identification

HFP AG>

HFRE Disable Sound_Enhancement Indication, ID: 0x0001

HFRE_Send_Terminating_Response (erOK) :: Res = 0

HFP AG>HFRE Speaker Gain Indication, ID: 0x0001, Speaker Gain 0x000A.

HFP AG>HFRE Call Line Identification Notification Activation Indication, ID: 0x0001, Enabled: TRUE.

HFP AG>HFRE Call Waiting Notification Activation Indication, ID: 0x0001, Enabled: TRUE.

HFP AG>

HFRE Response Hold Status Indication, ID: 0x0001

HFRE_Send_Incoming_Call_State (csNone) :: Res = 0

HFP AG>

HFRE_Current_Calls_List_Indication

HFRE_Send_Terminating_Response (erOK) :: Res = 0

5. Initiate audio connection by issuing the: ManageAudio <STATE> command which, by defaut chooses to uses Modified sub-band coding (MSBC).

   HFP AG>ManageAudio

   Usage: Audio [Release = 0, Setup = 1].

   Function Error.

   HFP AG>ManageAudio 1

   HFRE_Send_Select_Codec:: Codec = 2, Res = 0

   You will see the below output from the HFP server

   HFP AG>

   HFRE Codec Select Confirmation, ID: 0x0001 AcceptedCodec=2

   Setup WBS with Audio for ACL handle 0x0001

   HFRE_Setup_Audio_Connection: Function Successful.

   HFRE_Setup_Audio_Connection:: Res = 0

   HFP AG>

   HFRE Audio Connection Indication, ID: 0x0001, BDADDR=0x08DF1F99F8D0, Status: 0x0000.

6.3 Application Commands

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands

ServiceDiscovery

Description

The following function is responsible for issuing a Service Search Attribute Request to a Remote SDP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

The command requires two parameter. first parameter is the Inquiry Index and the second is Profile Index.

Command Call Examples

"ServiceDiscovery 1 12" Attempts to Open a Audio Gateway client port on inquiry index #1 and Profile Index Number #12 (Hands-free).

"ServiceDiscovery 3 12" Attempts to Open a Audio Gateway client port on inquiry index #3 and Profile Index Number #12 (Hands-free).

Possible Return Values

(-1) BTPS_ERROR_INVALID_PARAMETER (-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-40) BTPS_ERROR_MEMORY_ALLOCATION_ERROR (-54) BTPS_ERROR_EXPECTED_UUID_ENTRY (-64) BTPS_ERROR_INTERNAL_ERROR (-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

API Call

SDP_Service_Search_Attribute_Request(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam - 1)], 1, &SDPUUIDEntry, 1, &AttributeID, SDP_Event_Callback, (unsigned long)0)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI SDP_Service_Search_Attribute_Request(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, unsigned int NumberServiceUUID, SDP_UUID_Entry_t SDP_UUID_Entry[], unsigned int NumberAttributeListElements, SDP_Attribute_ID_List_Entry_t AttributeIDList[], SDP_Response_Callback_t SDP_Response_Callback, unsigned long CallbackParameter)

Description of API

The following function is responsible for issuing an SDP Service Search Attribute Request to the specified BD_ADDR. This function will return the result of the Service Search Attribute Request in the SDP Response Callback that is specified in the calling of this function. This function accepts as input, the Bluetooth Stack ID of the Bluetooth Protocol Stack that the SDP Client resides on, the Bluetooth Board Address to remotely connect to (the Remote SDP Server will reside on this BD_ADDR), the Number of Service UUID's that are to be searched for, the Servcee UUID's to actually search for, the Number of Entries in the Attribute List that are to be queried, the Attribute List to actually use in the Query, the SDP Response Callback Function, and the SDP Response Callback Function Callback Parameter. This function will return a positive, non-zero, return code if successful, or a negative return error code if there was an error. If this function is successful, the user can call the SDP_Cancel_Service_Request() function to cancel a the SDP Service Search Request prematurely.It should be noted that the Number of UUID Parameter must be at least one, and the Service UUID Parameter must point to a List of at least the Number of UUID's that have been specified. Finally, the BD_ADDR Parameter and the SDP_Reponse_Callback Parameter MUST be valid or the call to this function will be unsuccessful. It should also be noted that the Number of Attributes that are in the Attribute List must be at least one, and the Attribute ID List Parameter must point to a List of Attribute ID's that contains at least the Number of Attribute List Entries that have been specified. Finally, the BD_ADDR Parameter and the SDP_Reponse_Callback Parameter MUST be valid or the call to this function will be unsuccessful.

OpenAudioGatewayClient

Description

The following function is responsible for opening an Audio Gateway client port. This function returns zero on successful execution and a negative value on all errors.

Parameters

The command requires two parameter. first parameter is the Inquiry Index and the second is the remote device Port Number

Command Call Examples

"OpenAudioGatewayClient 1 1" Attempts to Open a Audio Gateway client port on inquiry index #1 and remote Port Number #1.

"OpenAudioGatewayClient 2 3" Attempts to Open a Audio Gateway client port on inquiry index #2 and remote Port Number #3.

Possible Return Values

(0) AG Client opened successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Open_Remote_Hands-free_Port(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam-1)], TempParam->Params[1].intParam, DEFAULT_AG_SUPPORTED_FEATURES, DEFAULT_CALL_HOLDING_SUPPORT, 0, NULL, HFRE_Event_Callback, (unsigned long)0)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Open_Remote_Hands-free_Port(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, unsigned int RemoteServerPort, unsigned long SupportedFeaturesMask, unsigned long CallHoldSupportMask, unsigned int NumberAdditionalIndicators, HFRE_Control_Indicator_Entry_t AdditionalSupportedIndicators[], HFRE_Event_Callback_t EventCallback, unsigned long CallbackParameter)

Description of API

The following function is responsible for Opening a Remote Hands-Free Port on the specified Remote Device. This function accepts the Bluetooth Stack ID of the Bluetooth Stack which is to open the HFRE Connection as the first parameter. The second parameter specifies the Board Address (NON NULL) of the Remote Bluetooth Device to connect with. The third parameter specifies the features the local Audio Gateway supports. The next parameter is a bit mask which specifies the Call Hold and Multiparty Handling Features that are supported. The fifth parameter to this function is the number of indicator names which appear in the list represented by the previous parameter. The next parameter is a list of additional indicators in which this Audio Gateway will support. If the Additional Indicators parameter is NULL and Number Additional Indicators is zero no additional parameters will be supported. The final two parameters specify the HFRE Event Callback function, and callback parameter, respectively, of the HFRE Event Callback that is to process any further interaction with the specified Remote Port (Opening Status, Close Status,etc). This function returns a non-zero, positive, value if successful, or a negative return error code if this function is unsuccessful. If this function is successful, the return value will represent the HFRE Port ID that can be passed to all other functions that require it. Once a Remote Hands-Free unit is opened, it can only be closed via a call to the HFRE_Close_Port() function (passing the return value from this function).

Note

The Mandatory Hands-Free Indicators (call, service, and call_setup) are automatically added to the list and need not be specified as additional indicators.

ManageAudio

Description

The following function is responsible for setting up or releasing an audio connection. This function returns zero on successful execution and a negative value on all errors.

Parameters

The Manage Audio command requires only one parameter to which is is an integer value that represents the ManageAudio mode. This value must be specified as 0 (for Release) or 1 (for Setup)

Command Call Examples

"ManageAudio 0" Attempts to Release the Audio Connection.

"ManageAudio 1" Attempts to Setup the Audio Connection.

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Setup_Audio_Connection(BluetoothStackID, CurrentClientPortID)

or HFRE_Release_Audio_Connection(BluetoothStackID, CurrentClientPortID)

API Prototype

int BTPSAPI HFRE_Setup_Audio_Connection(unsigned int BluetoothStackID, unsigned int HFREPortID)

or int BTPSAPI HFRE_Release_Audio_Connection(unsigned int BluetoothStackID, unsigned int HFREPortID)

Description of API

This function is responsible for Setting Up an Audio Connection between the Local and Remote Device. This function may be used by either an Audio Gateway or a Hands-Free unit for which a valid Service Level Connection Exists. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. This function returns zero if successful or a negative return error code if there was an error.

(or) This function is responsible for Releasing an Audio Connection which was previously established by the Remote Device or by a call to the HFRE_Setup_Audio_Connection() function. This function may be used by either an Audio Gateway or a Hands-Free unit. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. This function returns zero if successful or a negative return error code if there was an error.

UpdateControlIndicators

Description

The following function is responsible for updating the current state of the Control Indicators on the Remote Hands-Free unit. This function returns zero on successful execution and a negative value on all errors. indicators name : Call Status, Call Setup, Service availability, Signal indicator, Roam indicator, battchg indicator and Call Held

Parameters

The command requires two parameter. first parameter is the indicators name and the second is the value you want to set.

Command Call Examples

"UpdateControlIndicators Call Setup 1"

"UpdateControlIndicators Call Held 3"

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Update_Current_Control_Indicator_Status(BluetoothStackID, PortToUse, 1, &HFREIndicatorUpdate)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Update_Current_Control_Indicator_Status(unsigned int BluetoothStackID, unsigned int HFREPortID, unsigned int NumberUpdateIndicators, HFRE_Indicator_Update_t UpdateIndicators[])

Description of API

The following function is responsible for Updating the Current Control Indicator Status. This function may only be performed by Audio Gateways that have a valid Service Level Connection or by Audio Gateways that have received the etHFRE_Control_Indicator_Request_Indication event. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. The third parameter to this function is the number of name/value pairs that are present in the list. The final parameter to this function is a list of name/value pairs for the indicators to be updated. This function returns zero if successful or a negative return error code if there was an error.

CallWaiting

Note

This function will only work if call waiting feature is supported in the remote HF device.

Description

The following function is responsible for sending a Call Waiting Notification to the Remote Hands-Free unit. This function returns zero on successful execution and a negative value on all errors. If the remote device callwaiting feature is supported in the HF device than this will work.

Parameters

The Call Waiting command takes only one parameter, which is the phone number to be sent as part of this response.

Command Call Examples

"CallWaiting +9198787899889"

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Send_Call_Waiting_Notification(BluetoothStackID, CurrentClientPortID, TempParam->Params->strParam)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Send_Call_Waiting_Notification(unsigned int BluetoothStackID, unsigned int HFREPortID, char *PhoneNumber)

Description of API

This function is responsible for Sending Call Waiting Notifications to the Remote Device. This function may only be performed by Audio Gateways which have Call Waiting Notification Enabled and have a valid Service Level Connection. This function accepts as its first input parameter the HFRE Port ID. The final parameter is the Phone Number required as one of the parameters within this response. This parameter should be a pointer to a NULL terminated string (if specified) and must have a length less than HFRE_PHONE_NUMBER_LENGTH_MAXIMUM. This function returns zero if successful or a negative return error code if there was an error.

Note

It is valid to either pass a NULL for the PhoneNumber parameter of a blank string to specify that there is no phone number present.

SetVoiceRecognitionActivation

Description

The following function is responsible for deactivating Voice Recognition Activation on the Audio Gateway and for change the Voice Recognition Activation state on the Hands-free Unit. This function returns zero on successful execution and a negative value on all errors.

Parameters

The Voice Recognition command takes only one parameter, This value must be specified as 0 (for Deactivate) or 1 (to Activate).

Command Call Examples

"SetVoiceRecognitionActivation 0"

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Set_Remote_Voice_Recognition_Activation(BluetoothStackID, CurrentClientPortID, (Boolean_t)TempParam->Params->intParam)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Set_Remote_Voice_Recognition_Activation(unsigned int BluetoothStackID, unsigned int HFREPortID, Boolean_t VoiceRecognitionActive)

Description of API

This function is responsible for activation and deactivation of the Voice Recognition which resides on the Remote Audio Gateway when called by a Hands-Free unit. When called by an Audio Gateway this function is responsible for informing the remote Hands-Free unit of the current activation state of the local Voice Recognition function. This function may only be called by local devices that were opened with the Supported Feature Bit set for Voice Recognition. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. The final parameter is a Boolean flag specifying if this is a call to Activate or Deactivate this function on a Remote Audio Gateway, or to specify that Voice Recoginition is locally Activated or Deactivated to a Remote Hands-Free unit. When active the Voice Recognition function on the Audio Gateway is turned on, when inactive the Voice Recognition function on the Audio Gateway is turned off. This function returns zero if successful or a negative return error code if there was an error.

SetSpeakerGain

Description

The following function is responsible for setting the Speaker Gain on Remote Device. This function returns zero on successful execution and a negative value on all errors. If the remote audio volume control feature is supported in the HF device than this will work.

Parameters

The command requires only one parameter. It would be a integer value for the Speaker Gain.

Command Call Examples

"SetSpeakerGain 5"

"SetSpeakerGain 6"

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Set_Remote_Speaker_Gain(BluetoothStackID, CurrentClientPortID, TempParam->Params->intParam)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Set_Remote_Speaker_Gain(unsigned int BluetoothStackID, unsigned int HFREPortID, unsigned int SpeakerGain)

Description of API

This function is responsible for allowing synchronization and setting of the Remote Devices Speaker Gain. This function may only be performed if a valid Service Level Connection exists. When called by a Hands-Free unit this function is provided as a means to inform the Remote Audio Gateway of the current Speaker Gain value. When called by an Audio Gateway this function provides a means for the Audio Gateway to control the Speaker Gain of the Remote Hands-Free unit. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. The final parameter is the Speaker Gain to be sent to the Remote Device. The Speaker Gain Parameter *MUST* be between the values of HFRE_SPEAKER_GAIN_MINIMUM and HFRE_SPEAKER_GAIN_MAXIMUM. This function returns zero if successful or a negative return error code if there was an error.

SetMicroPhoneGain

Description

The following function is responsible for setting the Microphone Gain on Remote Device. This function returns zero on successful execution and a negative value on all errors. If the remote audio volume control feature is supported in the HF device than this will work.

Parameters

The command requires only one parameter. It would be a integer value for the MicroPhone Gain.

Command Call Examples

"SetMicroPhoneGain 5"

"SetMicroPhoneGain 6"

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Set_Remote_Microphone_Gain(BluetoothStackID, CurrentClientPortID, TempParam->Params->intParam)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Set_Remote_Microphone_Gain(unsigned int BluetoothStackID, unsigned int HFREPortID, unsigned int MicrophoneGain)

Description of API

This function is responsible for allowing synchronization and setting of the Remote Devices Microphone Gain. This function may only be performed if a valid Service Level Connection exists. When called by a Hands-Free unit this function is provided as a means to inform the Remote Audio Gateway of the current Microphone Gain value. When called by an Audio Gateway this function provides a means for the Audio Gateway to control the

Microphone Gain of the Remote Hands-Free unit. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. The final parameter is the Microphone Gain to be sent to the Remote Device. The Microphone Gain Parameter *MUST* be between the values of HFRE_MICROPHONE_GAIN_MINIMUM and HFRE_MICROPHONE_GAIN_MAXIMUM. This function returns zero if successful or a negative return error code if there was an error.

DisableRemoteSoundEnhancement

Note

This function may be performed only when a valid Service Level Connection exist but no Audio Connection exists.

Description

The following function is responsible for disabling Sound Enhancement on the Remote Device. This function returns zero on successful execution and a negative value on all errors. This function may be performed only when a valid Service Level Connection exist but no Audio Connection exists.

Parameters

t is not necessary to include parameters when using this command, A parameter will have no effect on the outcome of DisableRemoteSoundEnhancement.

Command Call Examples

"DisableRemoteSoundEnhancement"

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Disable_Remote_Echo_Cancelation_Noise_Reduction(BluetoothStackID, CurrentClientPortID)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Disable_Remote_Echo_Cancelation_Noise_Reduction(unsigned int BluetoothStackID, unsigned int HFREPortID)

Description of API

This function is responsible for Disabling Echo Cancelation and Noise Reduction on the Remote Device. This function may be performed by both the Hands-Free unit and the Audio Gateway for which a valid Service Level Connection exist but no Audio Connection exists. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. This function returns zero if successful or a negative return error code if there was an error.

Note

It is not possible to enable this feature once it has been disbled because the specification provides no means to re-enable this feature. This feature will remained disabled until the current Service Level Connection has been dropped.

SendCallerIdNotification

Description

The following function is responsible for sending a Call Line Identification Notification to the Remote Hands-Free unit. This function returns zero on successful execution and a negative value on all errors.

Parameters

The command requires only one parameter. It would be a CALLER id (phone number).

Command Call Examples

"SendCallerIdNotification +9198787899889"

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Send_Call_Line_Identification_Notification(BluetoothStackID, CurrentClientPortID, TempParam->Params->strParam)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Send_Call_Line_Identification_Notification(unsigned int BluetoothStackID, unsigned int HFREPortID, char *PhoneNumber)

Description of API

This function is responsible for sending Call Line Identification Notifications to the Remote device. This function may only be performed by Audio Gateways which have Call Line Identification Notification Enabled and have a valid Service Level Connection. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. The final parameter is the Phone Number required as one of the parameters within this response. This parameter should be a pointer to a NULL terminated string and its length *MUST* be between the values of HFRE_PHONE_NUMBER_LENGTH_MINIMUM and HFRE_PHONE_NUMBER_LENGTH_MAXIMUM. This function return zero if successful or a negative return error code if there was an error.

SetRingIndication

Description

The following function is responsible for sending an enable or disable In-Band Ring Indication to the Remote Hands-Free unit. This function returns zero on successful execution and a negative value on all errors.

Parameters

The command requires only one parameter. This value must be specified as 0 (for disable) or 1 (for enabling).

Command Call Examples

"SetRingIndication 1" for enabling In-Band Ring Indication

"SetRingIndication 0" for disable In-Band Ring Indication

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Enable_Remote_InBand_Ring_Tone_Setting(BluetoothStackID, CurrentClientPortID, TempParam->Params[0].intParam? TRUE : FALSE)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Enable_Remote_InBand_Ring_Tone_Setting(unsigned int BluetoothStackID, unsigned int HFREPortID, Boolean_t EnableInBandRing)

Description of API

This function is responsible for Enabling or Disabling In-Band Ring Tone Capabilities for the Local Device. This function may only be performed by Audio Gateways for which a valid Service Level Connection exists. This function may only be used to enable In-Ring Tone Capabilities if the Local Audio Gateway supports this feature. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. The final parameter is a Boolean flag specifying if this is a call to Enable or Disable this functionality. This function returns zero if successful or a negative return error code if there was an error.

RingIndication

Description

The following function is responsible for sending a Ring Indication to the Remote Hands-Free unit. This function returns zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of RingIndication.

Command Call Examples

"RingIndication" send ring indication.

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Ring_Indication(BluetoothStackID, CurrentClientPortID)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Ring_Indication(unsigned int BluetoothStackID, unsigned int HFREPortID)

Description of API

This function is responsible for sending a Ring Indication to the Remote Hands-Free unit. This function may only be performed by Audio Gateways for which a valid Service Level Connection exists. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. This function returns zero if successful or a negative return error code if there was an error.

SendIncomingCallState

DescriptionThe following function is responsible for sending the Response and Hold Command (+BTRH) to the remote device. This function returns zero on successful execution and a negative value on all errors.

Parameters

This command requires only one parameter to which is is an integer value that represents the incoming call state. This value must be specified as 0 (incoming call is put on hold), 1 (held call is accept) and 2(held call is reject).

Command Call Examples

"SendIncomingCallState 1" if there is a incoming call.

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Send_Incoming_Call_State(BluetoothStackID, CurrentClientPortID, (HFRE_Call_State_t) CallState)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Send_Incoming_Call_State(unsigned int BluetoothStackID, unsigned int HFREPortID, HFRE_Call_State_t CallState)

Description of API

The following function is responsible for sending information about the incoming call state. This function may only be performed by Audio Gateways that have a valid Service Level Connection. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. The final parameter to this function indicates the call state to set as part of this message. This function returns zero if successful or a negative return error code if there was an error.

CloseAgClient

Description

The following function is responsible for closing any open HFP ports. This function returns zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of Close.

Possible Return Values

(0) Client closed successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Close_Port(BluetoothStackID, CurrentClientPortID)

API Prototype

int BTPSAPI HFRE_Close_Port(unsigned int BluetoothStackID, unsigned int HFREPortID)

Description of API

The following function exists to close a HFRE Port that was previously opened by any of the following mechanisms:

- Successful call to HFRE_Open_Remote_Hands-free_Port() function.

- Successful call to HFRE_Open_Remote_Audio_Gateway_Port() function.

- Incoming open request (Hands-Free or Audio Gateway) which the server was opened with either the HFRE_Open_Hands-free_Server_Port() or the HFRE_Open_Audio_Gateway_Server_Port() functions.

This function accepts as input the Bluetooth Stack ID of the Bluetooth Stack which the Open HFRE Port resides and the HFRE Port ID (return value from one of the above mentioned Open functions) of the Port to Close. This function returns zero if successful, or a negative return value if there was an error. This function does NOT Un-Register a HFRE Server Port from the system, it ONLY disconnects any connection that is currently active on the Server Port. The HFRE_Close_Server_Port()function can be used to Un-Register the HFRE Server Port.

SendOperatorInfo

Note

This function should be performed When received a request to query the network operator selection from the remote Hands-free device.

Description

The following function is responsible for sending the command to send an Operator Selection Response to the Remote Hands-free Device. When received a request to query the remote network operator selection from the the remote Hands-free device. This function returns zero on successful execution and a negative value on all errors.

Parameters

The Send Operator info command requires only one parameter which is the current Network Operator.

Command Call Examples

"SendOperatorInfo airtel"

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Send_Network_Operator_Selection(BluetoothStackID, CurrentClientPortID, 1, TempParam->Params[0].strParam)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Send_Network_Operator_Selection(unsigned int BluetoothStackID, unsigned int HFREPortID, unsigned int NetworkMode, char *NetworkOperator)

Description of API

The following function is responsible for sending the network operator. This function may only be performed by Audio Gateways that have received a request to query the remote network operator selection. This function

accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. The third parameter to this function is the current Network Mode. The final parameter The final parameter is the current Network Operator. This parameter should be a pointer to a NULL terminated string (if specified) and must have a length less than HFRE_NETWORK_OPERATOR_LENGTH_MAXIMUM. This function returns zero if successful or a negative return error code if there was an error.

Note

It is valid to either pass a NULL for the NetworkOperator parameter of a blank string to specify that there is no Network Operator present.

SendSubNumber

Note

This function should be performed when received a request to query the subscriber number information from the the remote Hands-free device.

Description

The following function is responsible for sending the command to send a subscriber number to the Remote Hands-free Device. when received a request to query the subscriber number information from the remote Hands-free device. This function returns zero on successful execution and a negative value on all errors.

Parameters

The Send Subscriber Number command requires only one parameter which is the phone number to be sent as part of this response.

Command Call Examples

"SendSubNumber +9198787899889"

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Send_Subscriber_Number_Information(BluetoothStackID, CurrentClientPortID, TempParam->Params[0].strParam, 4, HFRE_DEFAULT_NUMBER_FORMAT, TRUE)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Send_Subscriber_Number_Information(unsigned int BluetoothStackID, unsigned int HFREPortID, char *PhoneNumber, unsigned int ServiceType, unsigned int NumberFormat, Boolean_t FinalEntry)

Description of API

The following function is responsible for sending Subscriber Number Information. This function may only be performed by Audio Gateways that have received a request to query the subscriber number information. This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. The third parameter to this function is the phone number to be sent as part of this response. The Phone Number Parameter String Length *MUST* be between the values of HFRE_PHONE_NUMBER_LENGTH_MINIMUM and HFRE_PHONE_NUMBER_LENGTH_MAXIMUM. The fourth parameter to this function is the Service type related to the specified Phone Number. The fifth paramter to this function is the Number Format to use for this number, The final parameter to this function is a boolean indicating if this is the last subscriber number information entry to be sent therefore requiring that an OK be sent as well. This function returns zero if successful or a negative return error code if there was an error

SendCallList

Note

This function should be performed when received a request to query the current calls list information from the the remote Hands-free device.

Description

The following function is responsible for sending the command(s) to send the Call Entry to a remote Hands-free Device. When received a request to query the remote current calls list from the remote Hands-free device. This function returns zero on successful execution and a negative value on all errors.

Parameters

The Send Call list command takes six parameter, they are Index, call direction, call status, call Mode, Multiparty, Phone Number (in the same order).

Command Call Examples

"SendCallList 0 1 1 0 0 5551212"

Possible Return Values

(0) Command sent successfully

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHFRE_ERROR_INVALID_PARAMETER

(-1001) BTHFRE_ERROR_NOT_INITIALIZED

(-1002) BTHFRE_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTHFRE_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTHFRE_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTHFRE_ERROR_INVALID_OPERATION

(-1006) BTHFRE_ERROR_INVALID_CODEC_ID

API Call

HFRE_Send_Current_Calls_List(BluetoothStackID, CurrentClientPortID, &CurrentCallListEntry, Final)

API Prototype

BTPSAPI_DECLARATION int BTPSAPI HFRE_Send_Current_Calls_List(unsigned int BluetoothStackID, unsigned int HFREPortID, HFRE_Current_Call_List_Entry_t *CurrentCallListEntry, Boolean_t FinalEntry)

Description of API

The following function is responsible for Sending Current Calls List Entries to the Remote Device. This function may only be performed by Audio Gateways that have received a request to query the remote current calls list.

This function accepts as its input parameters the Bluetooth Stack ID for which the HFRE Port ID is valid as well as the HFRE Port ID. The third parameter to this function is the current call list entry to be sent. The final parameter to this function is a boolean indicating if this is the last call list entry to be sent therefore requiring that an OK be sent as well. This function returns zero if successful or a negative return error code if there was an error.

Note

If the third parameter is specified as NULL, then the Final parameter *MUST* specify that this is the Final Entry. In this case, no call list entry will be sent, however, the terminating response will be sent. This function does not send the Phonebook Name as part of the Call List Entry. Use HFRE_Send_Current_Calls_List_With_Phonebook_Name if you wish to also send the Phonebook Name with the entry.

7 HID Demo Guide

7.1 Demo Overview

Note

The same instructions can be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms.

The human interface Device enables a Host to connect and control a HID Device. There are two roles defined in this profile, Host and Device. The first is the Host which sends control and report requests and the second is the Device which responds to the Host's requests. The Host is a Device that like a computer or tablet and the Device is a I/O Device like keyboard or mouse.

This application allows the user to use a console to use Bluetooth Low Energy (BLE) to establish connection between two BLE devices, control the Device, and get Reports and Protocols.

It is recommended that the user visits the kit setup Getting Started Guide for MSP430, Getting Started Guide for TIVA, Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as MSP-EXP430F5438 USB - Serial Port(COM x), Tiva Virtual COM Port (COM x), XDS110 Class Application/User UART (COM x) for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud (9600 for MSP430), 8, n, 1. Once connected, reset the Device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

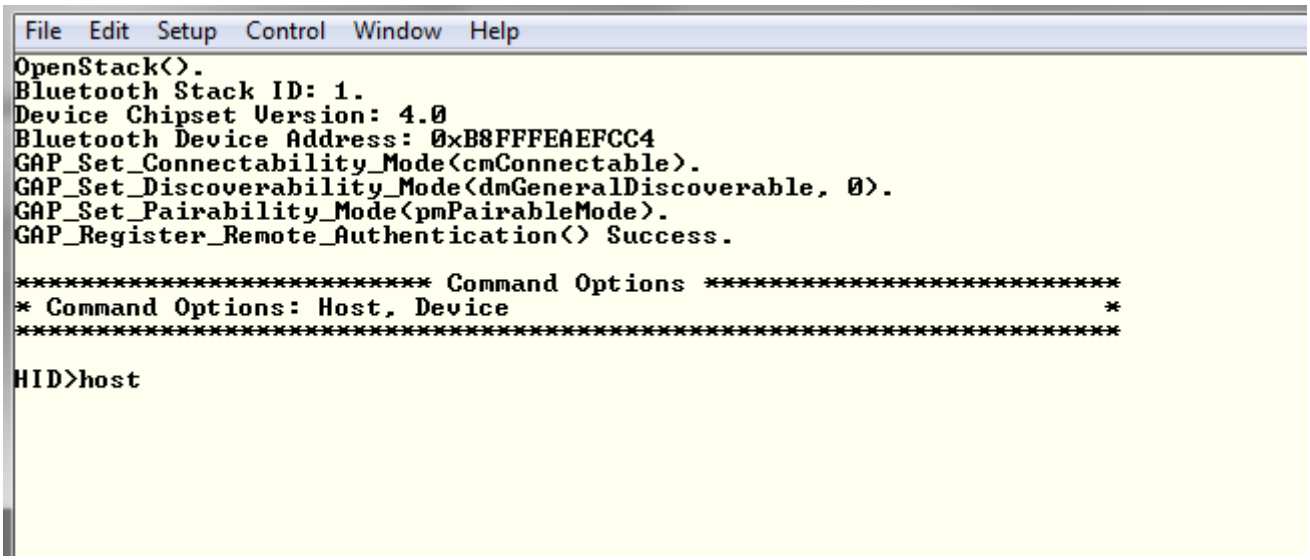

Figure 7-1. HID Demo Start Host Terminal

7.2 Demo Application

The demo application provides a description on how to use the demo application to connect two configured boards and communicate over bluetoothLE. The included application registers a custom service on a board when the stack is initialized.

Device 1 (Host/ HID Host) setup on the demo application

1. We will setup the first board as a Host. Perform the steps mentioned earlier in Running the Bluetooth Code section to initialize the application. Once initialized, note the Bluetooth address of the server. We will later use this to initiate a connection from the Device or Client.
2. On the Choose mode> prompt, enter Host.

3. You will see a list of all possible commands at this time for a Host. You can see this list at any time by entering Help at the Host> prompt.

Figure 7-2. HID Demo Register Server Terminal

4. At the Host> prompt, enter Inquiry This will initiate the Inquiry process. Once it is complete, you will get a list of all discovered devices.

```
HID>inquiry
Return Value is 0 GAP_Perform_Inquiry() SUCCESS.

HID>
GAP Inquiry Entry Result: 0xB8FFFEA92617.

HID>
GAP Inquiry Entry Result: 0x000272212389.

HID>
GAP Inquiry Entry Result: 0x30144A39DDB7.

HID>
GAP Inquiry Entry Result: 0x0007808031B7.

HID>
GAP Inquiry Entry Result: 0xBC0DA5F8D90E.

HID>
GAP Inquiry Entry Result: 0x0002724614C5.

HID>
GAP Inquiry Entry Result: 0x0007808031D1.

HID>
GAP Inquiry Entry Result: 0x00190E05483B.

HID>
GAP Inquiry Entry Result: 0xCDABDEADBEEF.

HID>
GAP Inquiry Entry Result: 0x001638392F2F.

HID>
GAP Inquiry Entry Result: 0x00190E0D47A3.

HID>
GAP Inquiry Entry Result: 0x7C8EE4001E72.

HID>
GAP Inquiry Entry Result: 0x000272D69F2D.

HID>
GAP Inquiry Entry Result: 0xB8FFFEAEFCC4.

HID>
GAP Inquiry Entry Result: 0x000272D69F2E.

HID>
GAP Inquiry Entry Result: 0x10BF48ED2C24.

HID>
GAP Inquiry Entry Result: 0x000272D69F33.

HID>
GAP Inquiry Entry Result: 0x000272D6A6E2.

HID>
GAP Inquiry Entry Result: 0x000780803205.

HID>
GAP Inquiry Entry Result: 0x000272D6A563.

HID>
GAP_Inquiry_Result: 20 Found.
GAP Inquiry Result: 1, 0xB8FFFEA92617.
GAP Inquiry Result: 2, 0x000272212389.
GAP Inquiry Result: 3, 0x30144A39DDB7.
GAP Inquiry Result: 4, 0x0007808031B7.
GAP Inquiry Result: 5, 0xBC0DA5F8D90E.
GAP Inquiry Result: 6, 0x0002724614C5.
GAP Inquiry Result: 7, 0x0007808031D1.
GAP Inquiry Result: 8, 0x00190E05483B.
GAP Inquiry Result: 9, 0xCDABDEADBEEF.
GAP Inquiry Result: 10, 0x001638392F2F.
GAP Inquiry Result: 11, 0x00190E0D47A3.
GAP Inquiry Result: 12, 0x7C8EE4001E72.
GAP Inquiry Result: 13, 0x000272D69F2D.
GAP Inquiry Result: 14, 0xB8FFFEAEFCC4.
GAP Inquiry Result: 15, 0x000272D69F2E.
```

Figure 7-3. HID Demo Inquiry Result Terminal

5. You can access this list any time by choosing DisplayInquiryList at the Host> prompt.

Device 2 (Client/HID Device) setup on the demo application

1. We will setup the second board as a Device. Perform the steps mentioned earlier in Running the Bluetooth Code section to initialize the application. On the Choose mode> prompt, enter Device.
2. You will see a list of all possible commands at this time for a Device. You can see this list at any time by entering Help at the Device> prompt.

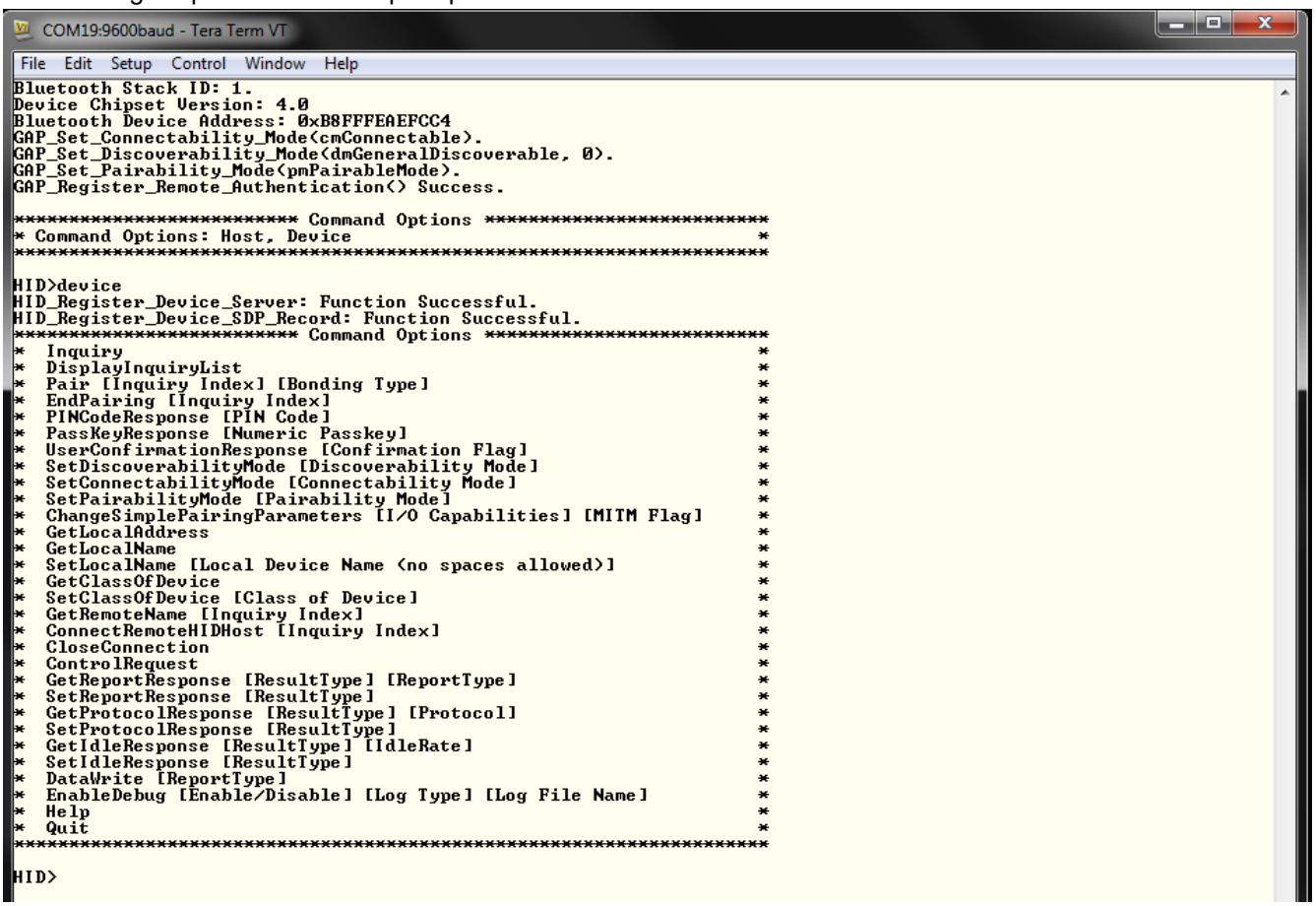

Figure 7-4. HID Demo SDP Record Terminal

Initiating connection from the HID Host

Note

When connecting for the first time, the connection should be initiated from the HID Host Device. If not the connection will be rejected as the HID Host will not allow a connection from an unknown HID Device.

1. Note the index number of the second board that was configured as a HID Device. [If the list is not on the screen, issue the DisplayInquiryList command on the client to display the list of discovered devices.]
2. Issue a ConnectRemoteHIDDevice <Inquiry Index> command from the Device.
3. Wait for HID Open confirmation.

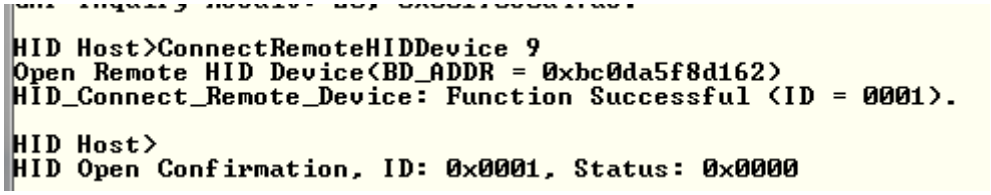

Figure 7-5. HID Demo Connect Device Terminal

Note

A status of 0x0000 means the connection was successful. Any other status implies that the connection did not succeed.

4. When a client successfully connects to a server, the server will see the open indication.

```
HID Open Indication, ID: 0x0001, Board: 0xB8FFFEA98279
```

Figure 7-6. HID Demo Open Indication Terminal

Initiating connection from the HID Device

Initiating a connection from the HID Device is the same as the procedure as the Host except that we run the inquiry(mentioned in Step 4) on the HID Device instead and issue a ConnectRemoteHIDHost <Inquiry Index> where the Inquiry Index is the number that corresponds to the BD-ADDR of the Host when we run the inquiry.

```
HID>ConnectRemoteHIDHost 12
Open Remote HID Host(BD_ADDR = 0xB8FFFEAEFCC4)
HID_Connect_Remote_Host: Function Successful (ID = 0001).

HID>
HID Open Confirmation, ID: 0x0001, Status: 0x0000
```

Figure 7-7. HID Demo Connect Device Terminal 2

Communication between Host and Device

1. Now that we have a connection established, the Host and Device can communicate with each other.
2. We can send a control operation from either the Host or the Device using the ControlRequest command. For the Host we issue the Controlrequest <parameter-number> command. The options for control request are 0= hcNop, 1= hcHardReset, 2= hcSoftReset, 3= hcSuspend, 4=hcExitSuspend, 5=hcVirtualCableUnplug. When we type Control Request 5 on the Host we get the following message.

```
HID>ControlRequest 5
HID_Control_Request: Function Successful.

HID>
HID Close Indication, ID: 0x0002
```

Figure 7-8. HID Demo Connect Device Terminal

On the Deviced side, we get a hcVirtualCableUnplug indication.

```
HID>
HID Control Indication, ID: 0x0001, Control Operation: hcVirtualCableUnplug

HID>
HID Close Indication, ID: 0x0001
```

Figure 7-9. HID Demo Control Indication Terminal

The Device controlRequest has no parameters. It does hcVirtualCableUnplug by defalt.

```
HID>
HID Control Indication, ID: 0x0003, Control Operation: hcVirtualCableUnplug

HID>
HID Close Indication, ID: 0x0003
```

Figure 7-10. HID Demo Control Indication Terminal 2

3. We can make Report Requests issuing the GetReportRequest command. It needs either 3 or 4 parameters. The first one is Size which is 0 for using the size of the Report or 1 for using a custom buffer size. The second is ReportType which is 0 for rtOther, 1 for rtInput, 2 for rtOutput, and 3 for rtFeature. The third is ReportID. IF we used a custom buffer for size in the first parameter we specify it here. For example, we send a Report Request with size of Report, rtInput and ReportID of 2. We get a Report Request Success indication.

```
HID>GetReportRequest 0 1 2
HID_Get_Report_Request: Function Successful.
```

Figure 7-11. HID Demo GetReportRequest

On the Device, we receive a get Report Indication with the Report Type, Id and size and Buffer size.

```
HID>
HID Get Report Indication, ID: 0x0003, ReportType: rtInput, ReportID: 2, Size: grSizeOfReport, BufferSize: 0
```

Figure 7-12. HID Demo Get Report Indication Terminal

The Device can respond to GetReportRequest using GetReportResponse. It needs the Result type (0 for rtSuccessful, 1 for rtNotReady, 2 for rtErrInvalidReportID, 3 for rtErrUnsupportedRequest, 4 for rtErrInvalidParameter, 5 for rtErrUnknown, 6 for rtErrFatal, 7 for rtData) and ReportType (0 for rtOther, 1 for rtInput, 2 for rtOutput, 3 for rtFeature) as parameters. For example we respond to the above rtInput request from the Host with rtData as Result type and rtInput as Report Type.

```
HID>ControlRequest 5
HID_Control_Request: Function Successful.

HID>
HID Close Indication, ID: 0x0002
```

Figure 7-13. HID Demo Get Report Response Terminal

The Host gets a report Confirmation back.

```
HID>
HID Get Report Confirmation, ID: 0x0004, Status: rtData, ReportType: rtInput, ReportLength: 4,
Report: 0x02 0x80 0x50 0x00
```

Figure 7-14. HID Demo Get Report Confirmation Terminal

4. We can run SetReportRequest from the Host. The only parameter it needs is the report type which is 0 for rtOther, 1 for rtInput, 2 for rtOutput, and 3 for rtFeature.

```
HID>SetReportRequest 1
HID_Set_Report_Request: Function Successful.
```

Figure 7-15. HID Demo SetReportRequest Terminal

The Device gets a Set Report Indication with the Report type.

```
HID>
HID Set Report Indication, ID: 0x0004, ReportType: rtInput, ReportLength: 4,
Report: 0x02 0x80 0x50 0x00
```

Figure 7-16. HID Demo SetReport Indication Terminal

The Device can respond to SetReportRequest by issuing the SetReportResponse command. The only parameter it needs is Result type (0 for rtSuccessful, 1 for rtNotReady, 2 for rtErrInvalidReportID, 3 for rtErrUnsupportedRequest, 4 for rtErrInvalidParameter, 5 for rtErrUnknown, 6 for rtErrFatal, 7 for rtData). For example we respond to the above rtInput Report request using rtSuccesful.

```
HID>SetReportResponse 0
HID_Set_Report_Response: Function Successful.
```

Figure 7-17. HID Demo Set Report Response Terminal

The Host receives a Set Report confirmation indication with the Result type.

```
HID>
HID Get Report Confirmation, ID: 0x0004, Status: rtData, ReportType: rtInput, ReportLength: 4,
Report: 0x02 0x80 0x50 0x00
```

Figure 7-18. HID Demo Set Report Confirmation Terminal

5. We can send a Protocol Request using GetProtocolRequest. It requires no parameters.

```
HID>GetProtocolRequest
HID_Get_Protocol_Request: Function Successful.
```

Figure 7-19. HID Demo GetProtocolRequest Terminal

The Device gets a Protocol Indication.

```
HID>
HID Get Protocol Indication, ID: 0x0003
```

Figure 7-20. HID Demo Get Protocol Indication Terminal

The Device can respond to the Protocol Request by issuing the GetProtocolResponse command. It requires two parameters, Result type (0 for rtSuccessful, 1 for rtNotReady, 2 for rtErrInvalidReportID, 3 for rtErrUnsupportedRequest, 4 for rtErrInvalidParameter, 5 for rtErrUnknown, 6 for rtErrFatal, and 7 for rtData) and Protocol (0 for ptBoot and 1 for ptReport). For example, we respond to the previous Request with rtData and ptBoot.

```
HID>GetProtocolResponse 7 0
HID_Get_Protocol_Response: Function Successful.

HID>
```

Figure 7-21. HID Demo Get Protocol Response Terminal 2

The Host gets a protocol confirmation with the Result type and Protocol.

```
HID>
HID Get Protocol Confirmation, ID: 0x0004, Status: rtData, Protocol: ptBoot
```

Figure 7-22. HID Demo Get Protocol Confirmation Terminal 2

6. We can issue SetProtocolRequest from the Host. The only parameter we need is the protocol(0 for ptBoot and 1 for ptReport). For example we send a request with ptReport.

```
HID>SetProtocolResponse 0
HID_Set_Protocol_Response: Function Successful.
```

Figure 7-23. HID Demo SetProtocolResponse Terminal 2

The Host gets a Set Protocol Indication along with the protocol. It can respond to it by issuing the SetProtocolResponse command which requires Result type as the parameter (0 for rtSuccessful, 1 for rtNotReady, 2 for rtErrInvalidReportID, 3 for rtErrUnsupportedRequest, 4 for rtErrInvalidParameter, 5 for rtErrUnknown, 6 for rtErrFatal, and 7 for rtData).

```
HID>
HID Set Protocol Indication, ID: 0x0001, Protocol: ptReport

HID>SetProtocolResponse 0
HID_Set_Protocol_Response: Function Successful.
```

Figure 7-24. HID Demo SetProtocolIndication Terminal 2

We get a Protocol Confirmation with the Result type in the Host.

```
HID>
HID Set Protocol Confirmation, ID: 0x0002, Status: rtSuccessful
```

Figure 7-25. HID Demo Set Protocol Confirmation Terminal 2

7. We can set the Idle request issuing the GetIdleRequest command. It requires no parameters.

```
HID>GetIdleRequest
HID_Get_Idle_Request: Function Successful.
```

Figure 7-26. HID Demo GetIdleRequest Terminal

The Device gets a GetIdleIndication.

```
HID>
HID Get Idle Indication, ID: 0x0001
```

Figure 7-27. HID Demo Get Idle Indication Terminal

It can respond to it with a GetIdleResponse which requires Result type (0 for rtSuccessful, 1 for rtNotReady, 2 for rtErrInvalidReportID, 3 for rtErrUnsupportedRequest, 4 for rtErrInvalidParameter, 5 for rtErrUnknown, 6 for rtErrFatal, and 7 for rtData) and Idle Rate as parameters. For example we respond with a Result type of rtData and Idle Rate of 50.

```
HID>GetIdleResponse 7 50
HID_Get_Idle_Response: Function Successful.
```

Figure 7-28. HID Demo Get Idle Response Terminal

On the Host we get a Idle Confirmation.

```
HID>
HID Get Idle Confirmation, ID: 0x0002, Status: rtData, IdleRate: 50
```

Figure 7-29. HID Demo Get Idle Confirmation Terminal 2

8. We can Set the Idle Rate using SetIdleRequest from the Host which requires Idle rate as the only parameter. For example we set the Idle rate to 50 from here.

```
HID>SetIdleRequest 50
HID_Set_Idle_Request: Function Successful.
```

Figure 7-30. HID Demo Set Idle Request Terminal 2

The Device receives a Set Idle Indication. It can respond to it using Set Idle Response which requires Result type (0 for rtSuccessful, 1 for rtNotReady, 2 for rtErrInvalidReportID, 3 for rtErrUnsupportedRequest, 4 for rtErrInvalidParameter, 5 for rtErrUnknown, 6 for rtErrFatal, and 7 for rtData) as its one parameter.

```
HID>
HID Set Idle Indication, ID: 0x0001, IdleRate: 50

HID>SetIdleResponse 0
HID_Set_Idle_Response: Function Successful.
```

Figure 7-31. HID Demo Set Idle Indication 2

We get a SetIdleConfirmation with the Result type on the Host

```
HID>
HID Set Idle Confirmation, ID: 0x0002, Status: rtSuccessful
```

Figure 7-32. HID Demo Set Idle Confirmation Terminal 2

9. We can write data between devices using DataWrite. The one parameter it needs is ReportType (0 for rtOther, 1 for rtInput, 2 for rtOutput, and 3 for rtFeature). We write data from the Device in this example.

```
HID>DataWrite 1
HID_Data_Write: Function Successful.
```

Figure 7-33. HID Demo DataWrite Terminal

The Host gets a Data indication.

```
HID>
HID Data Indication, ID: 0x0002, ReportType: rtInput, ReportLength: 4,
Report: 0x02 0x80 0x50 0x00
```

Figure 7-34. HID Demo Data Indication Terminal

7.3 Application Commands

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands

Host
ConnectRemoteHIDDevice

Description

The following function is responsible for Connecting to a Remote HID Device. This function returns zero on successful execution and a negative value on all errors.

Parameters

This command takes Inquiry Index number to work which can be found using the DisplayInquiryList command after an Inquiry has been completed.

Possible Return Values

(0)HID_Connect_Remote_Device: Function Successful

(-4) FUNCTION_ERROR

(-6)INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103)BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1004)BTHID_ERROR_INSUFFICIENT_RESOURCES

API Call

HID_Connect_Remote_Device(BluetoothStackID, InquiryResultList[(TempParam->Params->intParam-1)], &HIDConfiguration, HID_Event_Callback, 0)

API Prototype

int BTPSAPI HID_Connect_Remote_Device(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, HID_Configuration_t *HIDConfiguration, HID_Event_Callback_t EventCallback, unsigned long CallbackParameter)

Description of API

The following function is responsible for opening a connection to a Remote HID Device on the Specified Bluetooth Device. This function accepts as its first parameter the Bluetooth Stack ID of the Bluetooth Stack which is to open the HID Connection. The second parameter specifies the Board Address (NON NULL) of the Remote Bluetooth Device to connect with. The third parameter to this function is the HID Configuration Specification to be used in the negotiation of the L2CAP Channels associated with this Device Client. The final two parameters specify the HID Event Callback function and Callback Parameter, respectively, of the HID Event Callback that is to process any further events associated with this Device Client. This function returns a non-zero, positive, value if successful, or a negative return error code if this function is unsuccessful. If this function is successful, the return value will represent the HID ID that can be passed to all other functions that require it. Once a Connection is opened to a Remote Device it can only be closed via a call to the HID_Close_Connection() function (passing in the return value from a successful call to this function as the HID ID input parameter).

CloseConnection

DescriptionThe following function is responsible for closing any ongoing connection. This function returns zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of Close connection.

Possible Return Values

(0) HID_Close_Connection: Function Successful

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1004)BTHID_ERROR_INSUFFICIENT_RESOURCES

(-1005)BTHID_ERROR_INVALID_OPERATION

(-1006)BTHID_ERROR_REQUEST_OUTSTANDING

API Call

HID_Close_Connection(BluetoothStackID, HIDID)

API Prototype

int BTPSAPI HID_Close_Connection(unsigned int BluetoothStackID, unsigned int HIDID)

Description of API

The following function is responsible for closing a HID connection established through a connection made to a Registered Server or a connection that was made by calling either the HID_Open_Remote_Device() or HID_Open_Remote_Host() functions. This function accepts as input the Bluetooth Stack ID of the Bluetooth Protocol Stack that the HID ID specified by the Second Parameter is valid for. This function returns zero if successful, or a negative return error code if an error occurred. Note that if this function is called with the HID ID of a Local Server, the Server will remain registered but the connection associated with the specified HID ID will be closed.

ControlRequest

Description

The following function is responsible for sending a HID_CONTROL Transaction to the remote entity. This function returns zero on successful execution and a negative value on all errors.

Parameters

If the Device is a Host then it requires one parameter which is the Control Operation, 0= hcNop, 1= hcHardReset, 2= hcSoftReset, 3= hcSuspend, 4=hcExitSuspend, 5=hcVirtualCableUnplug. If the Device is not a Host then it is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of Control Request

Possible Return Values

(0) HID_Control_Request: Function Successful

(-4) FUNCTION_ERROR

(-6)INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005)BTHID_ERROR_INVALID_OPERATION

(-1006)BTHID_ERROR_REQUEST_OUTSTANDING

API Call

HID_Control_Request(BluetoothStackID, HIDID, (HID_Control_Operation_Type_t)((!IsHost)? (hcVirtualCableUnplug):(TempParam->Params->intParam)))

API Prototype

int BTPSAPI HID_Control_Request(unsigned int BluetoothStackID, unsigned int HIDID, HID_Control_Operation_Type_t ControlOperation)

Description of API

The following function is responsible for Sending a HID_CONTROL transaction to the remote side. This function accepts as input the Bluetooth Stack ID of the Bluetooth Stack which is to send the request and the HID ID for which the Connection has been established. The third parameter is the Control Operation that will be sent. This function returns zero if successful, or a negative return error code if there as an error.

Note

Control Channel Transfers normally consist of two phases, a Request by the Host and a Response by the Device. However, HID Control transactions require no Response phase. Note that HID Control Requests are not allowed while other transactions are being processed unless the Control Operation Type is hcVirtualCableUnplug which may be sent at any time.

GetReportRequest

Description

The following function is responsible for sending a GET_REPORT Transaction to the remote HID Device. This function returns zero on successful execution and a negative value on all errors.

Parameters

This function required three or four parameters. The first is size, 0 = grSizeOfReport, 1 = grUseBufferSize, the second is ReportType 0 = rtOther, 1 = rtInput, 2 = rtOutput, 3 = rtFeature, the third is ReportId. If the size parameter is 1, we need to specify a fourth parameter Buffersize.

Possible Return Values

(0) HID_Get_Report_Request: Function Successful

(-4) FUNCTION_ERROR

(-6)INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103)BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005)BTHID_ERROR_INVALID_OPERATION

API Call

HID_Get_Report_Request(BluetoothStackID, HIDID, (HID_Get_Report_Size_Type_t)TempParam->Params[0].intParam, (HID_Report_Type_Type_t)TempParam->Params[1].intParam, (Byte_t)(TempParam->Params[2].intParam), (Word_t)(TempParam->Params[3].intParam))

API Prototype

HID_Get_Report_Request(unsigned int BluetoothStackID, unsigned int HIDID, HID_Get_Report_Size_Type_t Size, HID_Report_Type_Type_t ReportType, Byte_t ReportID, Word_t BufferSize)

Description of API

The following function is responsible for Sending a GET_REPORT transaction to the remote Device. This function accepts as input the Bluetooth Stack ID of the Bluetooth Stack which is to send the request and the HID ID for which the Connection has been established. The third parameter is the descriptor that indicates how the Device is to determine the size of the buffer that the Host has allocated. The fourth parameter is the type of report requested. The fifth parameter is the Report ID determined by the Device's SDP record. Passing a zero for this parameter will indicate that this parameter is not used and will exclude the appropriate byte from the transaction payload. The fifth parameters use is based on the parameter passed as Size. If the Host indicates it has allocated a buffer of a size smaller than the report it is requesting, this parameter will be used as the size of the report returned. Otherwise, the appropriate bytes will not be included in the transaction payload. This function returns zero if successful or a negative return error code if there was an error.

Note

Control Channel transfers have two phases, a Request by the Host and a Response by the Device. Only ONE Host control channel Request shall be outstanding at a time. Reception of a HID Get Report Confirmation event indicates that a Response has been received and the Control Channel is now free for further Transactions.

SetReportRequest

Description

The following function is responsible for sending a SET_REPORT Transaction to the remote HID Device. This function returns zero on successful execution and a negative value on all errors.

Parameters

SetReportRequest uses one parameter, ReportType 0 = rtOther, 1 = rtInput, 2 = rtOutput, 3 = rtFeature.

Possible Return Values

(0)HID_Set_Report_Request: Function Successful. (-4) FUNCTION_ERROR

(-6)INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103)BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005)BTHID_ERROR_INVALID_OPERATION

(-1006)BTHID_ERROR_REQUEST_OUTSTANDING

API Call

HID_Set_Report_Request(BluetoothStackID, HIDID, (HID_Report_Type_Type_t)TempParam->Params[1].intParam, sizeof(GenericMouseReport), GenericMouseReport);

API Prototype

int BTPSAPI HID_Set_Report_Request(unsigned int BluetoothStackID, unsigned int HIDID, HID_Report_Type_Type_t ReportType, Word_t ReportPayloadSize, Byte_t *ReportDataPayload);

Description of API

The following function is responsible for sending a SET_REPORT request to the remote Device. This function accepts as input the Bluetooth Stack ID of the Bluetooth Stack which is to send the transaction and the HID ID for which the Connection has been established. The third parameter is the type of report being sent. Note that rtOther is an Invalid Report Type for use with this function. The final two parameters to this function are the Length of the Report Payload to send and a pointer to the Report Payload that will be sent. This function returns zero if successful or a negative return error code if there was an error.

Note

Control Channel transfers have two phases, a Request by the Host and a Response by the Device. Only ONE Host control channel Request shall be outstanding at a time. Reception of a HID Set Report Confirmation event indicates that a Response has been received and the Control Channel is now free for further Transactions.

GetProtocolRequest

Description

The following function is responsible for sending a GET_PROTOCOL Transaction to the remote HID Device. This function returns zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of GetProtocolRequest .

Possible Return Values

(0)HID_Get_Protocol_Request: Function Successful. (-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103)BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005)BTHID_ERROR_INVALID_OPERATION

(-1006)BTHID_ERROR_REQUEST_OUTSTANDING

API Call

HID_Get_Protocol_Request(BluetoothStackID, HIDID)

API Prototype

int BTPSAPI HID_Get_Protocol_Request(unsigned int BluetoothStackID, unsigned int HIDID);

Description of API

The following function is responsible for sending a GET_PROTOCOL transaction to the remote HID Device. This function accepts as input the Bluetooth Stack ID of the Bluetooth Stack which is to send the request and the HID ID for which the Connection has been established. This function returns a zero if successful or a negative return error code if there was an error.

Note

Control Channel transfers have two phases, a Request by the Host and a Response by the Device. Only ONE Host control channel request shall be outstanding at a time. Reception of a HID Get Protocol Confirmation event indicates that a response has been received and the Control Channel is now free for further Transactions.

SetProtocolRequest

Description

The following function is responsible for sending a SET_PROTOCOL Transaction to the remote HID Device. This function returns zero on successful execution and a negative value on all errors.

Parameters

SetProtocolRequest needs one parameter which is Protocol, 0= ptReport and 1=ptBoot.

Possible Return Values

(0) HID_Set_Protocol_Request: Function Successful (-4) FUNCTION_ERROR

(-6)INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103)BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005)BTHID_ERROR_INVALID_OPERATION

(-1006)BTHID_ERROR_REQUEST_OUTSTANDING

API Call

HID_Set_Protocol_Request(BluetoothStackID, HIDID, (HID_Protocol_Type_t)TempParam->Params[0].intParam)

API Prototype

int BTPSAPI HID_Set_Protocol_Request(unsigned int BluetoothStackID, unsigned int HIDID, HID_Protocol_Type_t Protocol)

Description of API

The following function is responsible for sending a SET_PROTOCOL transaction to the remote HID Device. This function accepts the Bluetooth Stack ID of the Bluetooth Stack which is to send the request and the HID ID for which the Connection has been established. The last parameter is the protocol to be set. This function returns a zero if successful or a negative return error code if there was an error.

Note

Control Channel transfers have two phases, a Request by the Host and a Response by the Device. Only ONE Host control channel Request shall be outstanding at a time. Reception of a HID Set Protocol Confirmation event indicates that a response has been received and the Control Channel is now free for further Transactions.

GetIdleRequest

Description

The following function is responsible for sending a GET_IDLE Transaction to the remote HID Device. This function returns zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of GetIdleRequest.

Possible Return Values

(0) HID_Get_Idle_Request: Function Successful. (-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103)BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005)BTHID_ERROR_INVALID_OPERATION

(-1006)BTHID_ERROR_REQUEST_OUTSTANDING

API Call

HID_Get_Idle_Request(BluetoothStackID, HIDID)

API Prototype

int BTPSAPI HID_Get_Idle_Request(unsigned int BluetoothStackID, unsigned int HIDID)

Description of API

The following function is responsible for sending a GET_IDLE transaction to the remote HID Device. This function accepts the Bluetooth Stack ID of the Bluetooth Stack which is to send the request and the HID ID for which the Connection has been established. This function returns a zero if successful or a negative return error code if there was an error.

Note

Control Channel transfers have two phases, a Request by the Host and a Response by the Device. Only ONE Host control channel request shall be outstanding at a time. Reception of a HID Get Idle Confirmation event indicates that a response has been received and the Control Channel is now free for further Transactions.

SetIdleRequest

Description

The following function is responsible for sending a SET_IDLE Transaction to the remote HID Device. This function returns zero on successful execution and a negative value on all errors.

Parameters

SetIdleRequest requires one Parameter which is Idlerate.

Possible Return Values

(0) HID_Set_Idle_Request: Function Successful

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1004)BTHID_ERROR_INSUFFICIENT_RESOURCES

(-1005)BTHID_ERROR_INVALID_OPERATION

(-1006)BTHID_ERROR_REQUEST_OUTSTANDING

API Call

HID_Set_Idle_Request(BluetoothStackID, HIDID, (Byte_t)TempParam->Params[0].intParam)

API Prototype

int BTPSAPI HID_Set_Idle_Request(unsigned int BluetoothStackID, unsigned int HIDID, Byte_t IdleRate)

Description of API

The following function is responsible for sending a SET_IDLE transaction to the remote HID Device. This function accepts the Bluetooth Stack ID of the Bluetooth Stack which is to send the request and the HID ID for which the Connection has been established. The last parameter is the Idle Rate to be set. The Idle Rate LSB is weighted to 4ms (i.e. the Idle Rate resolution is 4ms with a range from 4ms to 1.020s). This function returns a zero if successful or a negative return error code if there was an error.

Note

Control Channel transfers have two phases, a Request by the Host and a Response by the Device. Only ONE Host control channel request shall be outstanding at a time. Reception of a HID Set Idle Confirmation event indicates that a response has been received and the Control Channel is now free for further Transactions.

DataWrite

Description

The following function is responsible for sending a DATA Transaction on the Interrupt Channel to the remote entity. This function returns zero on successful execution and a negative value on all errors.

Parameters

It requires one parameter which is ReportType, 0 = rtOther, 1 = rtInput, 2 = rtOutput, 3 = rtFeature.

Possible Return Values

(0) HID_Control_Request: Function Successful

(-4) FUNCTION_ERROR

(-6)INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005)BTHID_ERROR_INVALID_OPERATION

API Call

HID_Data_Write(BluetoothStackID, HIDID, (HID_Report_Type_Type_t)TempParam->Params[0].intParam, sizeof(GenericMouseReport), GenericMouseReport)

API Prototype

int BTPSAPI HID_Data_Write(unsigned int BluetoothStackID, unsigned int HIDID, HID_Report_Type_Type_t ReportType, Word_t ReportPayloadSize, Byte_t *ReportDataPayload)

Description of API

The following function is responsible for sending Reports over the Interrupt Channel. This function accepts the Bluetooth Stack ID of the Bluetooth Stack which is to send the Report Data and the HID ID for which the Connection has been established. The third Parameter is the type of report being sent. The final two parameters are the Length of the Report Payload to send and a pointer to the Report Payload that will be sent. Note that rtOther and rtFeature are Invalid Report Types for use with this function. Also note that rtInput Reports must be sent from the Device to the Host, and that rtOutput Reports must be sent from the Host to the Device. This function returns a zero if successful or a negative return error code if there was an error.

Client
GetReportResponse

Description

The following function is responsible for sending a response for an outstanding GET_REPORT Transaction to the remote HID Host. This function returns zero on successful execution and a negative value on all errors.

Parameters

GetReportResponse requires two parameters, ResultType, 0= rtSuccessful 1= rtNotReady, 2= rtErrInvalidReportID, 3= rtErrUnsupportedRequest, 4= rtErrInvalidParameter, 5= rtErrUnknown, 6= rtErrFatal, 7= rtData and ReportType, 0 = rtOther, 1 = rtInput, 2 = rtOutput, 3 = rtFeature.

Possible Return Values

(0) HID_Get_Report_Response: Function Successful.

(-4) FUNCTION_ERROR

(-6)INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103)BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005)BTHID_ERROR_INVALID_OPERATION

(-1006)BTHID_ERROR_REQUEST_OUTSTANDING

API Call

HID_Get_Report_Response(BluetoothStackID, HIDID, (HID_Result_Type_t)TempParam->Params[0].intParam, (HID_Report_Type_Type_t)TempParam->Params[1].intParam, sizeof(GenericMouseReport), GenericMouseReport);

API Prototype

int BTPSAPI HID_Get_Report_Response(unsigned int BluetoothStackID, unsigned int HIDID, HID_Result_Type_t ResultType, HID_Report_Type_Type_t ReportType, Word_t ReportPayloadSize, Byte_t *ReportDataPayload);

Description of API

The following function is responsible for Sending the appropriate Response to an Outstanding GET_REPORT transaction. This function accepts the Bluetooth Stack ID of the Bluetooth Stack which is to send the response and the HID ID for which the Connection has been established. The third parameter to this function is the Result Type that is to be associated with this response. The rtSuccessful Result Type is Invalid for use with this

function. If the rtNotReady through rtErrFatal Result Statuses are used to respond, a HANDSHAKE response that has a Result Code parameter of the specified Error Condition is sent. If the ResultType specified is rtData, the GET_REPORT transaction is responded to with a DATA Response that has the Report (specified by the final parameter) as its Payload. The fourth parameter is the type of report being sent. Note that rtOther is an Invalid Report Type for use with this function. The final two parameters are the Length of the Report Payload to send and a pointer to the Report Payload that will be sent. This function returns a zero if successful, or a negative return error code if there was an error.

SetReportResponse

Description

The following function is responsible for sending a response for an outstanding SET_REPORT Transaction to the remote HID Host. This function returns zero on successful execution and a negative value on all errors.

Parameters

SetReportResponse requires one parameter which is ResultType, 0= rtSuccessful 1= rtNotReady, 2= rtErrInvalidReportID, 3= rtErrUnsupportedRequest, 4= rtErrInvalidParameter, 5= rtErrUnknown, 6= rtErrFatal, 7= rtData.

Possible Return Values

(0) HID_Set_Report_Response: Function Successful.

(-4) FUNCTION_ERROR

(-6)INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103)BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005)BTHID_ERROR_INVALID_OPERATION

API Call

HID_Get_Report_Response(BluetoothStackID, HIDID, (HID_Result_Type_t)TempParam->Params[0].intParam, (HID_Report_Type_Type_t)TempParam->Params[1].intParam, sizeof(GenericMouseReport), GenericMouseReport)

API Prototype

int BTPSAPI HID_Set_Report_Response(unsigned int BluetoothStackID, unsigned int HIDID, HID_Result_Type_t ResultType)

Description of API

The following function is responsible for Sending the appropriate Response to an Outstanding SET_REPORT transaction. This function accepts as input the Bluetooth Stack ID of the Bluetooth Stack which is to send the response and the HID ID for which the Connection has been established. The third parameter to this function is the Result Type that is to be associated with this response. The rtData Result Type is Invalid for use with this function. If the rtSuccessful through rtErrFatal Result Types are specified, this function responds to the SET_REPORT request with a HANDSHAKE response that has a Result Code parameter that matches the specified Result Type. This function returns zero if successful or a negative return error code if there was an error.

GetProtocolResponse

Description

The following function is responsible for sending a response for an outstanding GET_PROTOCOL Transaction to the remote HID Host. This function returns zero on successful execution and a negative value on all errors.

Parameters

GetProtocolResponse requires two parameters which are ResultType, 0= rtSuccessful, 1= rtNotReady, 2= rtErrInvalidReportID, 3= rtErrUnsupportedRequest, 4= rtErrInvalidParameter, 5= rtErrUnknown, 6= rtErrFatal, 7= rtData and Protocol, 0= ptReport and 1=ptBoot.

Possible Return Values

(0) HID_Get_Protocol_Response: Function Successful.

(-4) FUNCTION_ERROR

(-6)INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103)BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1004)BTHID_ERROR_INSUFFICIENT_RESOURCES

(-1005)BTHID_ERROR_INVALID_OPERATION

API Call

HID_Get_Protocol_Response(BluetoothStackID, HIDID, (HID_Result_Type_t)TempParam->Params[0].intParam, (HID_Protocol_Type_t)TempParam->Params[1].intParam)

API Prototype

int BTPSAPI HID_Get_Protocol_Response(unsigned int BluetoothStackID, unsigned int HIDID, HID_Result_Type_t ResultType, HID_Protocol_Type_t Protocol)

Description of API

The following function is responsible for Sending the appropriate Response to an Outstanding GET_PROTOCOL transaction. This function accepts the Bluetooth Stack ID of the Bluetooth Stack which is to send the response and the HID ID for which the Connection has been established. The third parameter to this function is the Result Type that is to be associated with this response. The rtSuccessful Result Type is Invalid for use with this function.If the rtNotReady through rtErrFatal Result Types are specified, this function will respond to the GET_PROTOCOL request with a HANDSHAKE response that has a Result Code parameter of the specified Error Condition. If the ResultType specified is rtData, the GET_PROTOCOL transaction is responded to with a DATA Response that has the Protocol type specified as the final parameter as its Payload. This function returns zero if successful or a negative return error code if there was an error.

SetProtocolResponse

(0) HID_Set_Protocol_Response: Function Successful.

(-4) FUNCTION_ERROR

(-6)INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103)BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005)BTHID_ERROR_INVALID_OPERATION

(-1006)BTHID_ERROR_REQUEST_OUTSTANDING

API Call

HID_Set_Protocol_Response(BluetoothStackID, HIDID, (HID_Protocol_Type_t)TempParam->Params[0].intParam)

API Prototype

int BTPSAPI HID_Set_Protocol_Response(unsigned int BluetoothStackID, unsigned int HIDID, HID_Result_Type_t ResultType);

Description of API

The following function is responsible for Sending the appropriate Response to an Outstanding SET_PROTOCOL transaction. This function accepts the Bluetooth Stack ID of the Bluetooth Stack which is to send the response and the HID ID for which the Connection has been established. The third parameter to this function is the Result Type that is to be associated with this response. The rtData Result Type is Invalid for use with this function. If the rtSuccessful through rtErrFatal Result Types are specified then this function will respond to the SET_PROTOCOL Transaction with a HANDSHAKE response that has a Result Code parameter that matches the specified Result Type. This function returns zero if successful, or a negative return error code if there was an error.

GetIdleResponse

Description

The following function is responsible for sending a response for an outstanding GET_IDLE Transaction to the remote HID Host. This function returns zero on successful execution and a negative value on all errors.

Parameters

GetIdleResponse requires two parameters, ResultType, 0= rtSuccessful, 1= rtNotReady, 2= rtErrInvalidReportID, 3= rtErrUnsupportedRequest, 4= rtErrInvalidParameter, 5= rtErrUnknown, 6= rtErrFatal, 7= rtData and IdleRate.

Possible Return Values

(0) HID_Set_Idle_Response: Function Successful.

(-4) FUNCTION_ERROR

(-6)INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103)BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005)BTHID_ERROR_INVALID_OPERATION

API Call

HID_Get_Idle_Response(BluetoothStackID, HIDID, (HID_Result_Type_t)TempParam->Params[0].intParam, (Byte_t)TempParam->Params[1].intParam)

API Prototype

int BTPSAPI HID_Get_Idle_Response(unsigned int BluetoothStackID, unsigned int HIDID, HID_Result_Type_t ResultType, Byte_t IdleRate)

Description of API

The following function is responsible for Sending the appropriate Response to an Outstanding GET_IDLE transaction. This function accepts the Bluetooth Stack ID of the Bluetooth Stack which is to send the response and the HID ID for which the Connection has been established. The third parameter to this function is the Result Type that is to be associated with this response. The rtSuccessful Result Type is Invalid for use with this function. If the rtNotReady through rtErrFatal Result Types are specified, then this function will respond to the GET_IDLE Transaction with a HANDSHAKE response that has a Result Code parameter of the specified Error Condition. If the ResultType specified is rtData the GET_IDLE transaction is responded to with a DATA Response that has the Idle Rate specified as the final parameter as its Payload. This function returns zero if successful, or a negative return error code if there was an error.

SetIdleResponse

Description

The following function is responsible for sending a response for an outstanding SET_IDLE Transaction to the remote HID Host. This function returns zero on successful execution and a negative value on all errors.

Parameters

SetIdleResponse requires one parameter which is ResultType, 0= rtSuccessful, 1= rtNotReady, 2= rtErrInvalidReportID, 3= rtErrUnsupportedRequest, 4= rtErrInvalidParameter, 5= rtErrUnknown, 6= rtErrFatal, 7= rtData.

Possible Return Values

(0) HID_Get_ Idle _Response: Function Successful.

(-4) FUNCTION_ERROR

(-6)INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103)BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000)BTHID_ERROR_INVALID_PARAMETER

(-1001)BTHID_ERROR_NOT_INITIALIZED

(-1002)BTHID_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005)BTHID_ERROR_INVALID_OPERATION

(-1006)BTHID_ERROR_REQUEST_OUTSTANDING

API Call

HID_Set_Idle_Response(BluetoothStackID, HIDID, (HID_Result_Type_t)TempParam->Params[0].intParam)

API Prototype

int BTPSAPI HID_Set_Idle_Response(unsigned int BluetoothStackID, unsigned int HIDID, HID_Result_Type_t ResultType)

Description of API

The following function is responsible for Sending the appropriate Response to an Outstanding SET_IDLE transaction. This function accepts the Bluetooth Stack ID of the Bluetooth Stack which is to send the response and the HID ID for which the Connection has been established. The third parameter to this function is the Result Type that is to be associated with this response. The rtData Result Type is Invalid for use with this function. If the rtSuccessful through rtErrFatal Result Types are specified, then this function will respond to the SET_IDLE Transaction with a HANDSHAKE response that has a Result Code parameter that matches the specified Result Type. This function returns zero if successful, or a negative return error code if there was an error.

8 HSP Demo Guide

8.1 Demo Overview

Note

The same instructions can be used to run this demo on the MSP432 or STM32F4 Platforms.

The Headset profile (HSP) is used to connect a headset or speakerphone with a mobile device or used to connect a Audio gateway with headset device to provide basic control and voice connections. The Headset profile supports two roles, Headset and Audio Gateway. This document demonstrates how to use both roles of the profile.

It is recommended that the user visits the kit setup Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as **XDS110 Class Application/User UART (COM x)** for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud, 8, n, 1. Once connected, reset the device using Reset S3 button (For MSP432) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

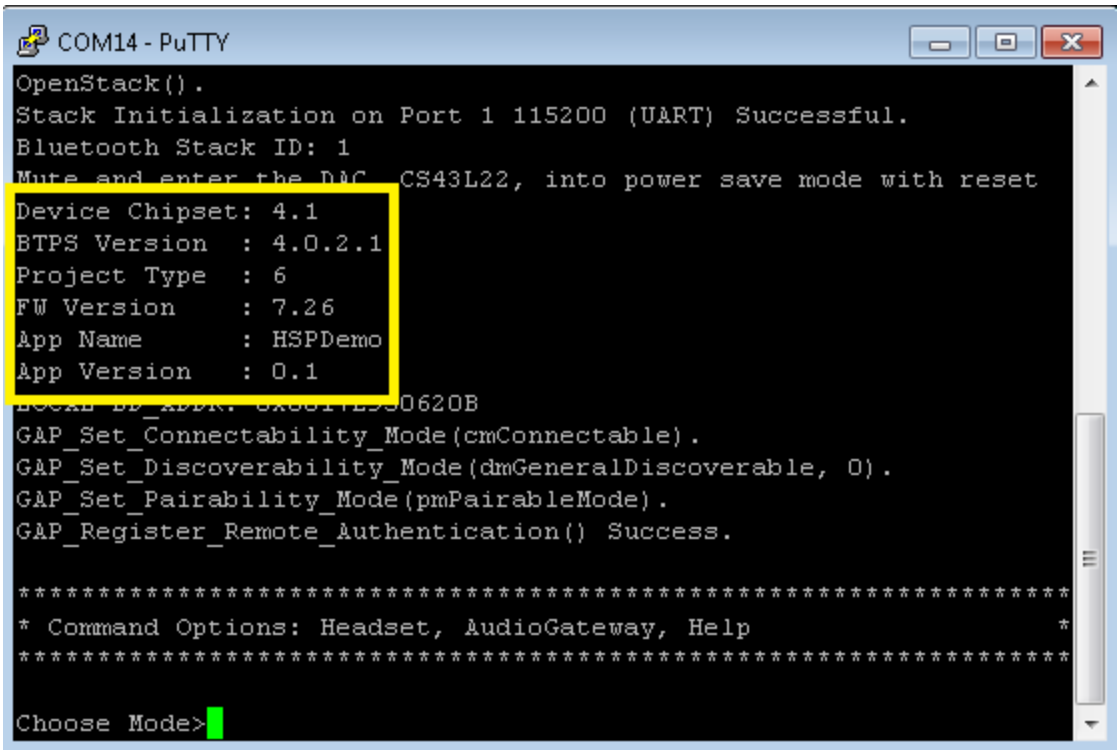

Figure 8-1. HSP Demo Start Screen

Note

The information shown in the yellow square holds the FW, BTPS and application versions for future use.

8.2 Demo Application

This section provides a description of how to use the demo application to connect smart phone over Bluetooth HSP profile, the same for the second smart phone. Bluetooth HSP is a simple Client-Server connection process with one side, the Client, operating in the Audio-Gateway role and the other, the Server, operating in the

Handsfree role. We will setup the boards as a Handsfree Server and use an android phone as the Client. Once connected, we can use the STM3240G-EVAL board as headset, with audio connected to the earphone jack.

Headset role

Server setup on the demo application

1. After initialization of the application we need to choose our role, this section will describe the **Headset** role, issue the **Headset** command in order to choose this role. After selecting the role you will be able to see the commands for this role.

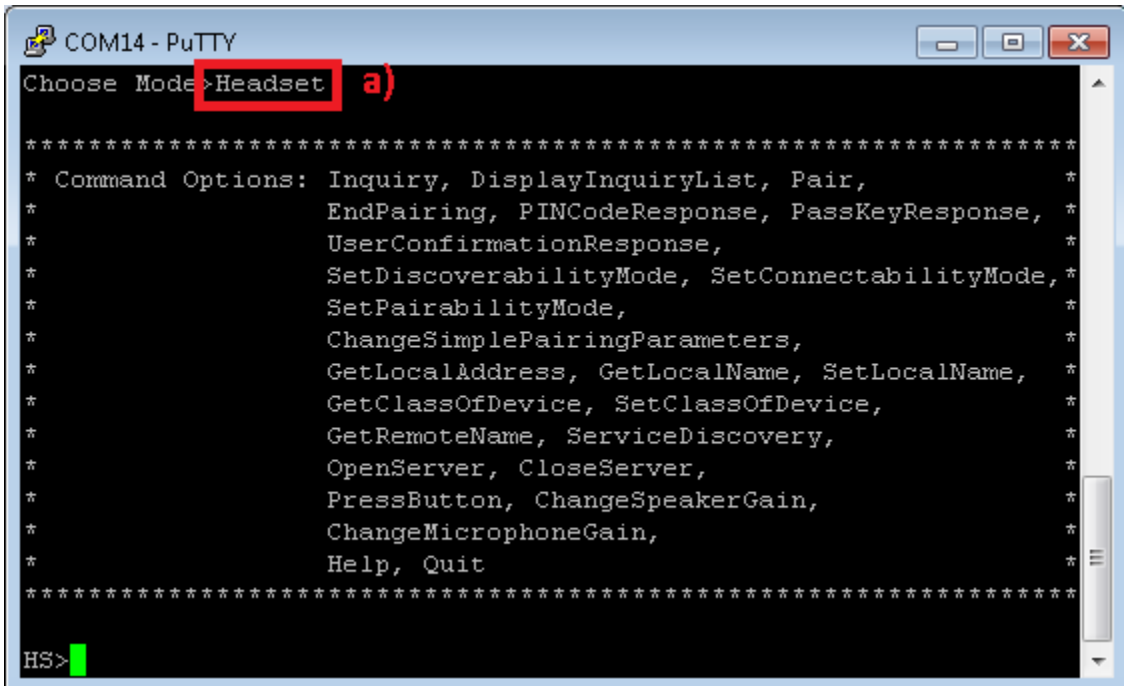

Figure 8-2. HSP Demo Headset Role

2. Optional: Give a name for the STM3240G-EVAL board issuing the **SetLocalName** command. In our example we give it a name of **hspserver**. The default application name is **HSPDemo**

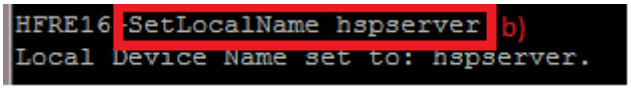

Figure 8-3. HSP Demo Set Name Print

3. Open a HSPServer by isssuing the **OpenServer** command. Below we use OpenServer to open the port.

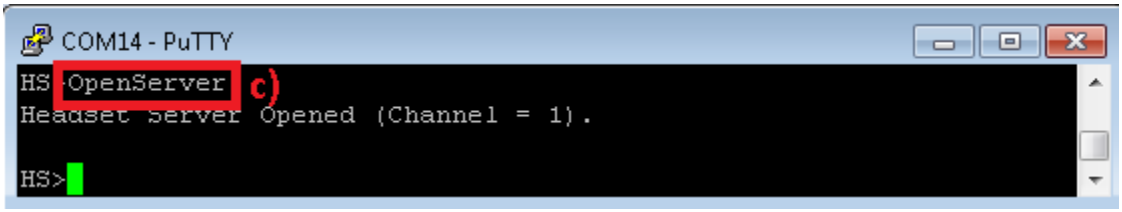

Figure 8-4. HSP Demo Open Server

Client setup and device discovery

4. Open the bluetooth settings menu on the android phone **Settings->Bluetooth**. The menu should look similar to the picture below in step 5.
5. Hit on **Search for devices**. The phone should begin looking for other bluetooth devices.

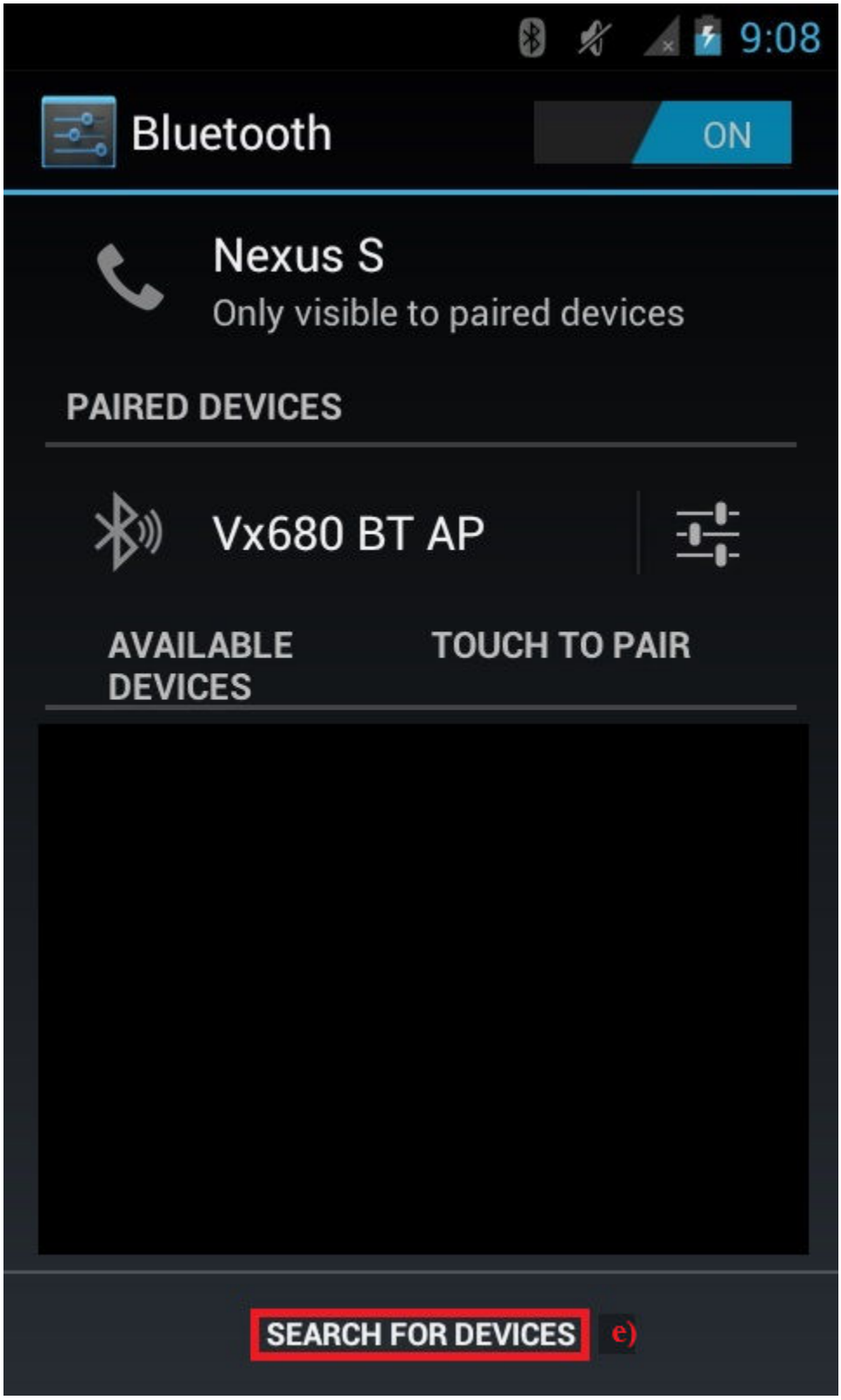

Figure 8-5. HSP Demo Bluetooth Settings

6. The Demo device should appear like shown below in the picture with the given name from section **b**. or the default name **HSPDemo**. Click on the device name to begin pairing.

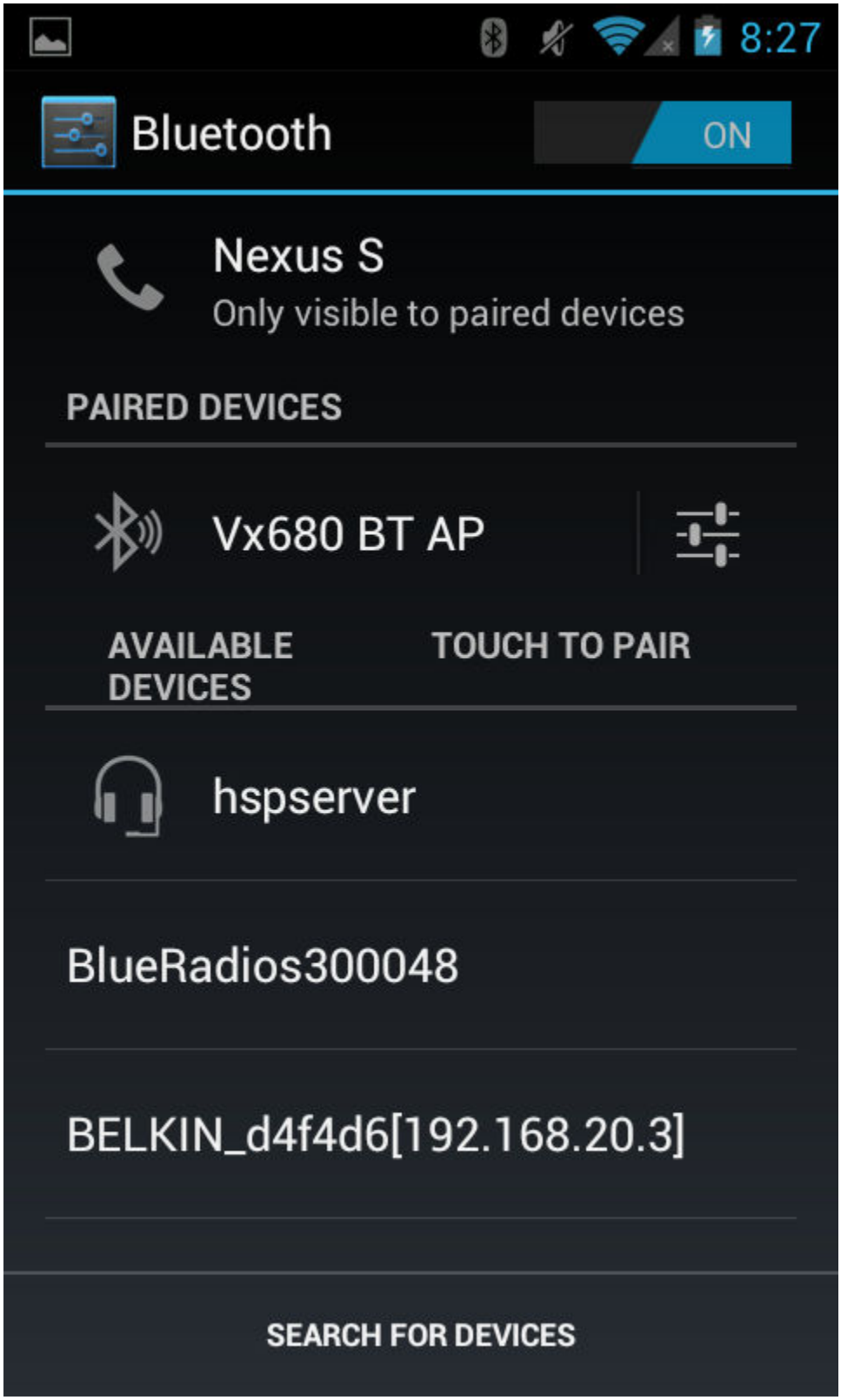

Figure 8-6. HSP Demo Bluetooth Search

7. After the devices are paired (In legacy pairing the Android will prompt for four digits code and then the terminal prompts for **PINCodeResponse** that should be answered with **PINCodeResponse <Four digit code>**, the device should show connected on the phone side and print **Open Service Level Connection Indication** on the terminal .

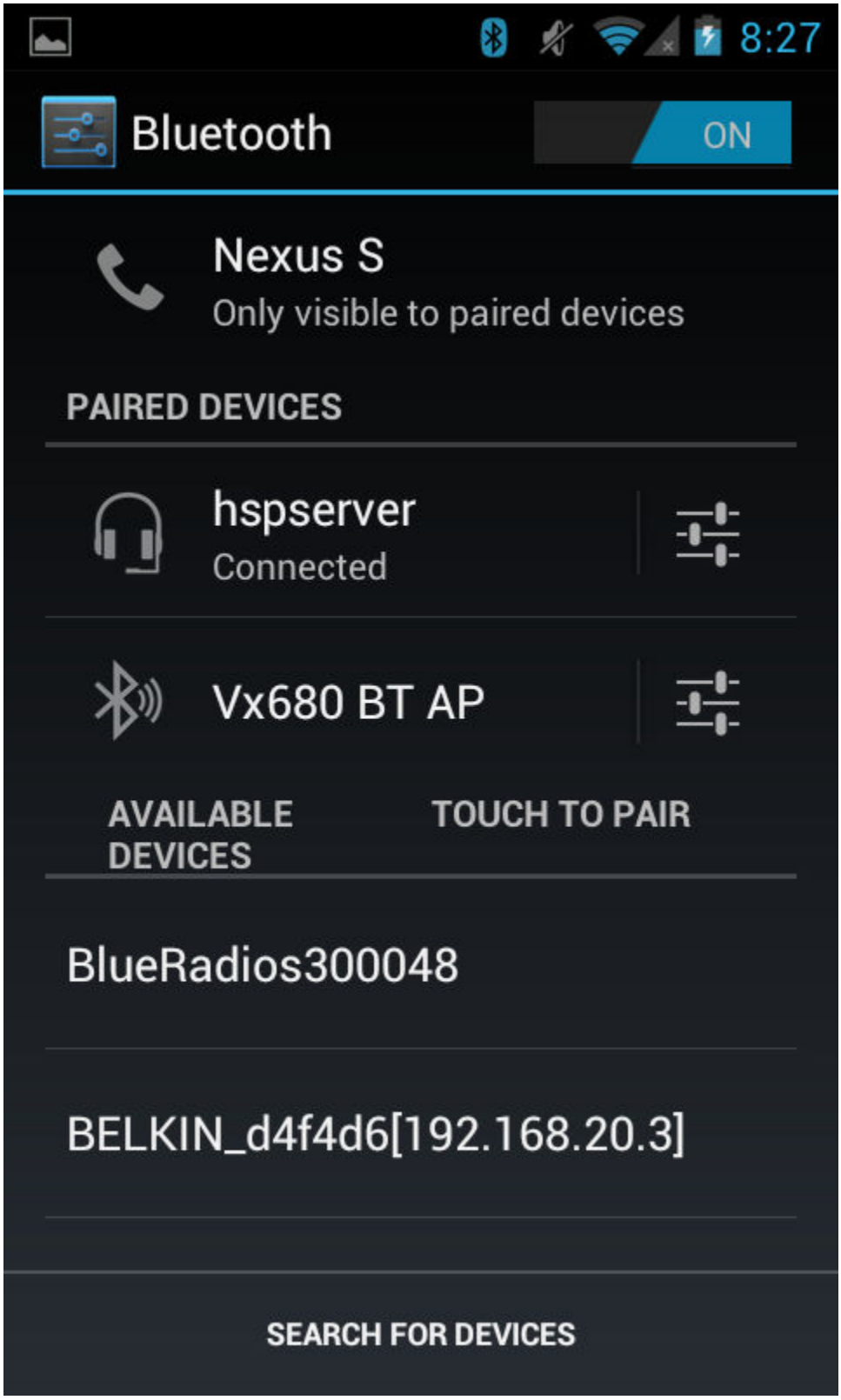

Figure 8-7. HSP Demo Bluetooth Connected

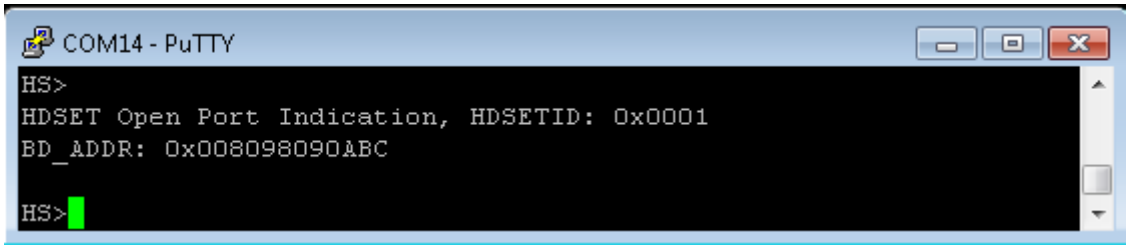

Figure 8-8. HSP Demo Connection

8. To Answer an Incoming Call or hang up an active call use **PressButton** Command.

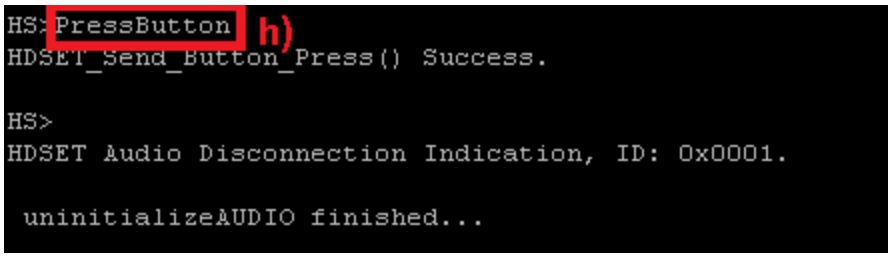

Figure 8-9. HSP Demo Answer Call

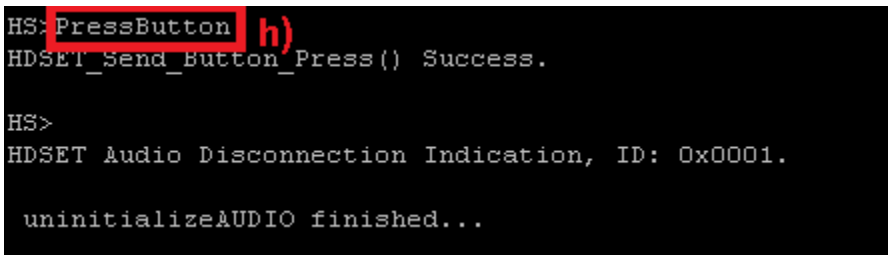

Figure 8-10. HSP Demo Hang up Call

9. To Close the **HSPserver**, issue the **CloseServer <port number>** command.

Figure 8-11. HSP Demo Close Server

Audio Gateway role

Note

The following instructions connect two boards running the HSP profile as a Headset and Audio Gateway.

Server setup on the demo application

1. After initialization of the application on the first board, we need to choose our role, this section will describe the **Headset** role, issue the **Headset** command in order to choose this role. After selecting the role you will be able to see the commands for this role.

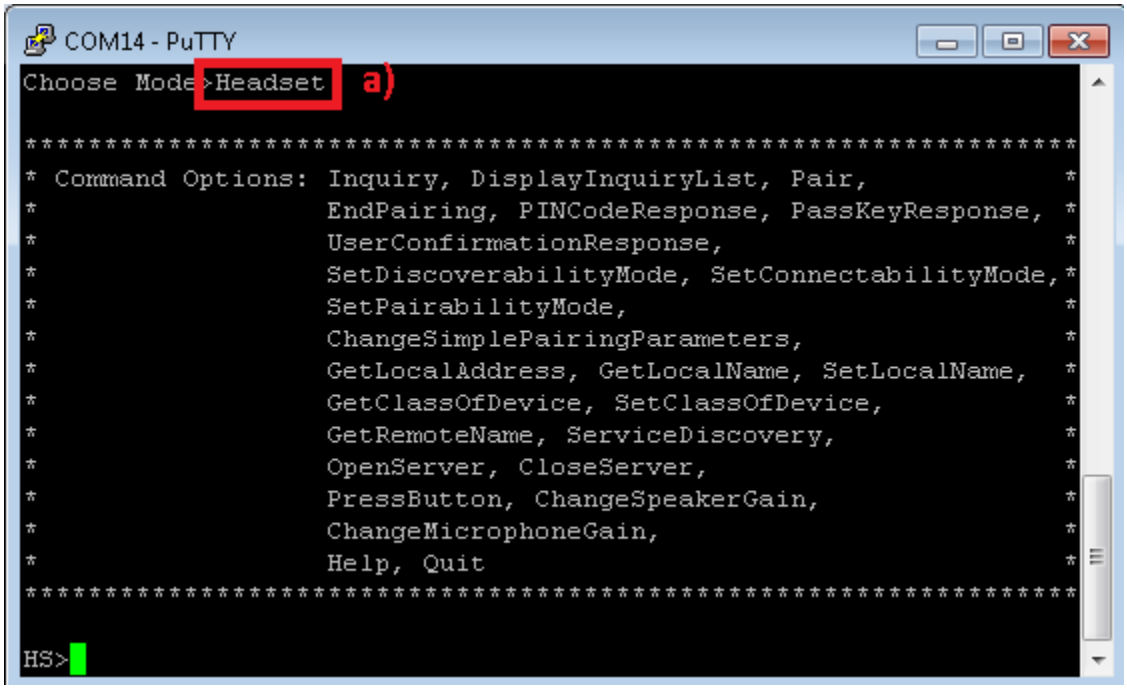

Figure 8-12. HSP Demo Headset Role

2. Optional: Give a name for the STM3240G-EVAL board issuing the **SetLocalName** command. In our example we give it a name of **hspserver**. The default application name is **HSPDemo**.

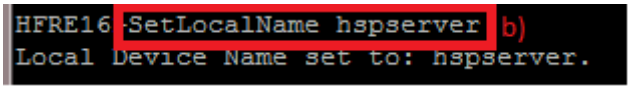

Figure 8-13. HSP Demo Set Name Print

3. Open a HSPServer by isssuing the **OpenServer** command. Below we use OpenServer to open the port.

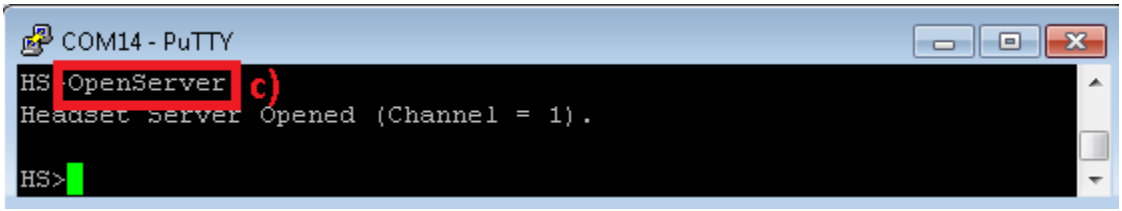

Figure 8-14. HSP Demo Open Server

Client setup and device discovery

4. After initialization of the application on the second board, we need to choose our role, this section will describe the **Audio Gateway** role. Issue the **AudioGateway** command in order to choose this role. After selecting the role you will be able to see the commands for this role.

```
**********************************************************************
* Command Options: Headset, AudioGateway, Help                       *
**********************************************************************

Choose Mode>audiogateway

**********************************************************************
* Command Options: Inquiry, DisplayInquiryList, Pair,                *
*                  EndPairing, PINCodeResponse, PassKeyResponse,     *
*                  UserConfirmationResponse,                         *
*                  SetDiscoverabilityMode, SetConnectabilityMode,    *
*                  SetPairabilityMode,                               *
*                  ChangeSimplePairingParameters,                    *
*                  GetLocalAddress, GetLocalName, SetLocalName,      *
*                  GetClassOfDevice, SetClassOfDevice,               *
*                  GetRemoteName, ServiceDiscovery,                  *
*                  OpenClient, CloseClient,                          *
*                  RingIndication, ChangeSpeakerGain,                *
*                  ChangeMicrophoneGain,                             *
*                  ManageAudio,                                      *
*                  Help, Quit                                        *
**********************************************************************
```

Figure 8-15. HSPD Demo Audio Gateway Audio Role

5. In order to open the **Client** port, we need to find the device we want to connect to, to do so we issue the **Inquiry** command to start scanning for nearby devices.

```
AG>inquiry
Return Value is 0 GAP_Perform_Inquiry() SUCCESS.

AG>
GAP Inquiry Entry Result: 0xDC53606A062A.

AG>
GAP Inquiry Entry Result: 0xE8B1FC980B7E.

AG>
GAP Inquiry Entry Result: 0x88C255D1D645.

AG>
GAP Inquiry Entry Result: 0x244B03F712D3.

AG>
GAP Inquiry Entry Result: 0xB0B448F4883D.

AG>
GAP_Inquiry_Result: 5 Found.
GAP Inquiry Result: 1, 0xDC53606A062A.
GAP Inquiry Result: 2, 0xE8B1FC980B7E.
GAP Inquiry Result: 3, 0x88C255D1D645.
GAP Inquiry Result: 4, 0x244B03F712D3.
GAP Inquiry Result: 5, 0xB0B448F4883D.
```

Figure 8-16. HSP Demo Audio Gateway Display Inquiry

6. After the Inquiry command has finished a list of found devices will be printed to the console. Note that we can retrieve the list again by issuing the **DisplayInquiryList** command.
7. After we found the device we need to issue the **OpenClient** command to open the **Client** port.

```
AG>openclient 3 1
HDSET_Open_Remote_Headset_Port() was successful.

AG>
HDSET Open Port Confirmation, HDSETID: 0x0001, Status 0x0000.
```

Figure 8-17. HSP Demo Open Client

8. You should receive an **HDSET Open Remote Headset Port** indication on the **Client** and **Server** consoles:

```
AG>
HDSET Open Port Confirmation, HDSETID: 0x0001, Status 0x0000.
```

Figure 8-18. HSP Demo Client Connection

The **Server** console also prints out the connected BD_ADDR.

```
HS>
HDSET Open Port Indication, HDSETID: 0x0001
BD_ADDR: 0xB0B448F49D74
```

Figure 8-19. HSP Demo Server Connection

9. You must now **ring** the Server device in order to begin the audio streaming process. This can be accomplished by issuing the **RingIndication** command on the **Client**.

```
AG>ringindication
HDSET_Ring_Indication() Success.
```

Figure 8-20. HSP Demo Start Connection

10. The **Server** will receive a **HDSET Ring Indication**.

```
HS>
HDSET Ring Indication, ID: 0x0001.
 Answer with respond command: PressButton
```

Figure 8-21. HSP Demo Ring Indication Notification

11. To Answer the Incoming Call or hang up an active call issue the **PressButton** command from the **Server**.

```
HS>pressbutton
HDSET_Send_Button_Press() Success.

HS>
HDSET Audio Connection Indication, ID: 0x0001.
```

Figure 8-22. HSP Demo Server Press Button

12. The **Client** will receive both a **HDSET Button Pressed** and **HDSET Audio Connection** indications.

```
AG>
HDSET Button Pressed Indication, ID: 0x0001, Connection Present: 0x0001.

AG>
HDSET Audio Connection Indication, ID: 0x0001.
```

Figure 8-23. HSP Demo Button Pressed Indication

13. To Close the **HSPserver**, issue the **CloseServer <port number>** command.

Example: Audio gateway with a commercial headset

This demonstrates setting up the client to connect to a commercial headset.

1. This section will describe the **Audio Gateway** role. Issue the **AudioGateway** command in order to choose this role. After selecting the role you will be able to see the commands for this role.

```
OpenStack().
Stack Initialization on Port 1 115200 (UART) Successful.
Bluetooth Stack ID: 1
Device Chipset: 4.1
BTPS Version  : 4.0.3.0
Project Type  : 6
FW Version    : 7.26
App Name      : HSPDemo
App Version   : 0.1
LOCAL BD_ADDR: 0xB0B448F49D74
GAP_Set_Connectability_Mode(cmConnectable).
GAP_Set_Discoverability_Mode(dmGeneralDiscoverable, 0).
GAP_Set_Pairability_Mode(pmPairableMode).
GAP_Register_Remote_Authentication() Success.

******************************************************************
* Command Options: Headset, AudioGateway, Help                   *
******************************************************************

Choose Mode>AudioGateway

******************************************************************
* Command Options: Inquiry, DisplayInquiryList, Pair,            *
*                  EndPairing, PINCodeResponse, PassKeyResponse, *
*                  UserConfirmationResponse,                     *
*                  SetDiscoverabilityMode, SetConnectabilityMode,*
*                  SetPairabilityMode,                           *
*                  ChangeSimplePairingParameters,                *
*                  GetLocalAddress, GetLocalName, SetLocalName,  *
*                  GetClassOfDevice, SetClassOfDevice,           *
*                  GetRemoteName, ServiceDiscovery,              *
*                  OpenClient, CloseClient,                      *
*                  RingIndication, ChangeSpeakerGain,            *
*                  ChangeMicrophoneGain,                         *
*                  ManageAudio,                                  *
*                  Help, Quit                                    *
******************************************************************
```

2. In order to open the **Client** port, we need to find the device we want to connect to, to do so we issue the **Inquiry** command to start scanning for nearby devices.

```
AG>Inquiry
Return Value is 0 GAP_Perform_Inquiry() SUCCESS.
```

3. After the Inquiry command has finished a list of found devices will be printed to the console.

Note

We can retrieve the list again by issuing the DisplayInquiryList command.

```
AG>
GAP Inquiry Entry Result: 0x244B03F712D3.

AG>
GAP Inquiry Entry Result: 0x340286605044.

AG>
GAP Inquiry Entry Result: 0x000DFD4072EF.

AG>
GAP_Inquiry_Result: 3 Found.
GAP Inquiry Result: 1, 0x244B03F712D3.
GAP Inquiry Result: 2, 0x340286605044.
GAP Inquiry Result: 3, 0x000DFD4072EF.
```

4. You can verify the device you want to connect to by issuing the **GetRemoteName <Inquiry Index>** command.

```
AG>GetRemoteName 3
GAP_Query_Remote_Device_Name: Function Successful.

AG>
GAP Remote Name Result: BD_ADDR: 0x000DFD4072EF.
GAP Remote Name Result: Motorola S10-HD.
```

5. Discover services of the remote HFP server by issuing the **ServiceDiscovery 3 11**, command to get the port number.

Note

The port ID on the remote Hands free device is 0x02 (The Unsigned int), from the Attribute ID 0x0004. This port ID is used in the following OpenAudioGatewayClient command as its second parameter after being converted to its decimal equivilant (10).

```
AG>ServiceDiscovery
Usage: SERVICEDISCOVERY [Inquiry Index] [Profile Index] [16/32 bit UUID (Manual only)].

   Profile Index:
       0) Manual (MUST specify 16/32 bit UUID)
       1) L2CAP
       2) Advanced Audio
       3) A/V Remote Control
       4) Basic Imaging
       5) Basic Printing
       6) Dial-up Networking
       7) FAX
       8) File Transfer
       9) Hard Copy Cable Repl.
      10) Health Device
      11) Headset
      12) Audio gateway
      13) HID
      14) LAN Access
      15) Message Access
      16) Object Push
      17) Personal Area Network
      18) Phonebook Access
      19) SIM Access
      20) Serial Port
```

```
      21) IrSYNC

Function Error.

AG>ServiceDiscovery 3 11
SDP_Service_Search_Attribute_Request(Headset) Success.

AG>
SDP Service Search Attribute Response Received (Size = 0x0010)
Service Record: 1:
Attribute ID 0x0000
Type: Unsigned Int = 0x00010001
Attribute ID 0x0001
Type: Data Element Sequence
Type: UUID_16 = 0x1108
Type: UUID_16 = 0x1203
Attribute ID 0x0004
Type: Data Element Sequence
Type: Data Element Sequence
Type: UUID_16 = 0x0100
Type: Data Element Sequence
Type: UUID_16 = 0x0003
Type: Unsigned Int = 0x02
Attribute ID 0x0006
Type: Data Element Sequence
Type: Unsigned Int = 0x656E
Type: Unsigned Int = 0x006A
Type: Unsigned Int = 0x0100
Attribute ID 0x0009
Type: Data Element Sequence
Type: Data Element Sequence
Type: UUID_16 = 0x1108
Type: Unsigned Int = 0x0100
Attribute ID 0x0100
Type: Text String = Headset
Attribute ID 0x0302
Type: Boolean = TRUE
```

6. After we found the device we need to issue the **OpenClient** command to open the **Client** port.

```
AG>OpenClient
Usage: Open [Inquiry Index] [RFCOMM Server Port].
Function Error.

AG>OpenClient 3 2
HDSET_Open_Remote_Headset_Port() was successful.
```

7. You should receive an **HDSET Open Remote Headset Port** indication on the **Client** console:

```
AG>
atPINCodeRequest: 0x000DFD4072EF

Respond with the command: PINCodeResponse

AG>PINCodeResponse 0000
GAP_Authentication_Response(), Pin Code Response Success.

AG>
atLinkKeyCreation: 0x000DFD4072EF
Link Key: 0x19CDEF146AF8A709A1C93BAA99A6E8EE
Link Key Stored locally.

AG>
HDSET Open Port Confirmation, HDSETID: 0x0001, Status 0x0000.

AG>
HDSET Speaker Gain Indication, ID: 0x0001, Gain: 0x0005.

AG>
HDSET Microphone Gain Indication, ID: 0x0001, Gain: 0x000A.
```

8. You must now **ring** the Server device in order to begin the audio streaming process. This can be accomplished by issuing the **RingIndication** command on the **Client**.

```
AG>RingIndication
HDSET_Ring_Indication() Success.
```

9. Next we will attempt to setup and release the audio connection on the current port.

```
AG>ManageAudio
Usage: Audio [Release = 0, Setup = 1].
Function Error.

AG>ManageAudio 1
HDSET_Setup_Audio_Connection: Function Successful.

AG>
HDSET Audio Connection Indication, ID: 0x0001.


AG>ManageAudio 0
HDSET_Release_Audio_Connection: Function Successful.

AG>RingIndication
HDSET_Ring_Indication() Success.
```

8.3 Application Command

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands

Headset Profile Commands

OpenServer

Description

The following function is responsible for opening a Serial Port Server on the Local Device. This function opens the Serial Port Server on the specified RFCOMM Channel. This function returns the opened port number (1-31) if successful, or a negative return value if an error occurred.

Parameters

None.

Possible Return Values

(1-32) HSP port opened successfully

(-8) INVALID_STACK_ID_ERROR

(-9) UNABLE_TO_REGISTER_SERVER

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000) BTHDSET_ERROR_INVALID_PARAMETER

(-1001) BTHDSET_ERROR_NOT_INITIALIZED

(-1002) BTHDSET_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1004) BTHDSET_ERROR_INSUFFICIENT_RESOURCES

API Call

HDSET_Open_Audio_Gateway_Server_Port(BluetoothStackID, LOCAL_SERVER_CHANNEL_ID, HDSET_Event_Callback, (unsigned long)0)

API Prototype

int BTPSAPI HDSET_Open_Audio_Gateway_Server_Port(unsigned int BluetoothStackID, unsigned int ServerPort, HDSET_Event_Callback_t EventCallback, unsigned long CallbackParameter)

Description of API

The following function is responsible for Opening an Audio Gateway Server on the specified Bluetooth SPP Serial Port. This function accepts as input the Bluetooth Stack ID of the Bluetooth Stack Instance to use for the Audio Gateway Server, the Local Serial Port Server Number to use, and the HDSET Event Callback

function (and parameter) to associate with the specified Headset Port. The ServerPort parameter *MUST* be between SPP_PORT_NUMBER_MINIMUM and SPP_PORT_NUMBER_MAXIMUM. This function returns a positive, non-zero, value if successful or a negative return error code if an error occurs. A successful return code will be a HDSET Port ID that can be used to reference the Opened HDSET Port in ALL other functions in this module except for the HDSET_Register_Headset_SDP_Record() function which is specific to a Headset Server NOT an Audio Gateway. Once a Server HDSET Port is opened, it can only be Un-Registered via a call to the HDSET_Close_Server_Port() function (passing the return value from this function). The HDSET_Close_Port() function can be used to Disconnect a Client from the Server Port (if one is connected, it will NOT Un-Register the Server Port however).

CloseServer

Description

The following function is responsible for closing a Serial Port Server that was previously opened via a successful call to the OpenServer() function. If the last Server is closed, the function also unregisters the SDP record. This function returns zero if successful or a negative return error code if there was an error.

Parameters

It is not necessary to include parameters when using this command.

Possible Return Values

(0) HSP Server closed successfully

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000)

(-1001) BTHDSET_ERROR_NOT_INITIALIZED

(-1002) BTHDSET_ERROR_INVALID_BLUETOOTH_STACK_ID

API Call

HDSET_Close_Server_Port(BluetoothStackID, HDSServerID)

API Prototype

int BTPSAPI HDSET_Close_Server_Port(unsigned int BluetoothStackID, unsigned int HDSETPortID)

Description of API

The following function is responsible for Un-Registering a HDSET Port Server (which was Registered by a successful call to either the HDSET_Open_Headset_Server_Port() or the HDSET_Open_Audio_Gateway_Server_Port() function). This function accepts as input the Bluetooth Stack ID of the Bluetooth Protocol Stack that the HDSET Port specified by the Second Parameter is valid for. This function returns zero if successful, or a negative return error code if an error occurred (see BTERRORS.H). Note that this function does NOT delete any SDP Service Record Handles.

PressButton

Description

The following function is responsible for issuing a Press Button Command for a Headset (either Accept or End a Call). This function returns zero if successful or a negative return error code if an error occurs.

Parameters

It is not necessary to include parameters when using this command.

Possible Return Values

(0) command sent successfully

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1001) BTHDSET_ERROR_NOT_INITIALIZED

(-1002) BTHDSET_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005) BTHDSET_ERROR_INVALID_OPERATION

API Call

HDSET_Send_Button_Press(BluetoothStackID, ((ServerConnected)?HDSServerID:HDSClientID))

API Prototype

int BTPSAPI HDSET_Send_Button_Press(unsigned int BluetoothStackID, unsigned int HDSETPortID)

Description of API

The following function is responsible for sending a Button Press to a remote Audio Gateway. This function accepts the Bluetooth Stack ID of the Bluetooth Stack which has received the HDSET Connection Request and the HDSET Port ID of the Headset to send the request on. This function returns a zero if successful, or a negative return error code if there was an error

Note

This function should be used instead of the:
HDSET_Accept_Incoming_Call()
or
HDSET_End_Call() functions.
The reason is that the above two functions imply a call state. Since the actual call state is handled via the Audio Gateway, the Headset does not have any mechanism to actually determine the call state. Because of this, this function will simply issue the Button Press and let the Audio Gateway decide how to process the request.

ChangeSpeakerGain

Description

The following function is responsible for sending a Speaker gain (Volume) Change Command to the Remote Connection. This function returns zero if successful or a negative return error code if an error occurs.

Parameters

One Parameter - Number Between 0-15

Possible Return Values

(0) command sent successfully

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHDSET_ERROR_INVALID_PARAMETER

(-1001) BTHDSET_ERROR_NOT_INITIALIZED

(-1002) BTHDSET_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005) BTHDSET_ERROR_INVALID_OPERATION

API Call

HDSET_Set_Speaker_Gain(BluetoothStackID, ((ServerConnected)?HDSServerID:HDSClientID), TempParam->Params[0].intParam)

API Prototype

int BTPSAPI HDSET_Set_Speaker_Gain(unsigned int BluetoothStackID, unsigned int HDSETPortID, unsigned int SpeakerGain)

Description of API

The following function is provided to allow the local entity a mechanism of notifying the Remote entity (either Headset OR Audio Gateway) that the Speaker Gain has changed. This function accepts as input the Bluetooth Stack ID of the Bluetooth Stack which the HDSET Port ID (second parameter) is valid for, the HDSET Port ID, and the new Speaker Gain Setting. This function returns zero if successful or a negative return error code if there was an error. The Speaker Gain Parameter *MUST* be between the values of HDSET_SPEAKER_GAIN_MINIMUM and HDSET_SPEAKER_GAIN_MAXIMUM.

ChangeMicrophoneGain

Description

The following function is responsible for sending a Change Microphone Gain Command to the Remote Connection. This function returns zero if successful or a negative return error code if an error occurs.

Parameters

One Parameter - Number Between 0-15

Possible Return Values

(0) Command sent successfully

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTHDSET_ERROR_INVALID_PARAMETER

(-1001) BTHDSET_ERROR_NOT_INITIALIZED

(-1002) BTHDSET_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005) BTHDSET_ERROR_INVALID_OPERATION

API Call

HDSET_Set_Microphone_Gain(BluetoothStackID, ((ServerConnected)?HDSServerID:HDSClientID), TempParam->Params[0].intParam)

API Prototype

int BTPSAPI HDSET_Set_Microphone_Gain(unsigned int BluetoothStackID, unsigned int HDSETPortID, unsigned int MicrophoneGain)

Description of API

The following function is provided to allow the local entity a mechanism of notifying the Remote entity (either Headset OR Audio Gateway) that the Microphone Gain has changed. This function accepts as input the Bluetooth Stack ID of the Bluetooth Stack which the HDSET Port ID (second parameter) is valid for, the HDSET Port ID, and the new Microphone Gain Setting. This function returns zero if successful or a negative return error code if there was an error. The Microphone Gain Parameter *MUST* be between the values of HDSET_MICROPHONE_GAIN_MINIMUM and HDSET_MICROPHONE_GAIN_MAXIMUM.

OpenClient

Description

The following function is responsible for initiating a connection with a Remote Headset or Audio Gateway Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

Two Parameters, First one is the Inquiry index, the Second is the RFCOMM Server Port.

Possible Return Values

(0) Command sent successfully

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000) BTHDSET_ERROR_INVALID_PARAMETER

(-1001) BTHDSET_ERROR_NOT_INITIALIZED

(-1002) BTHDSET_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1004) BTHDSET_ERROR_INSUFFICIENT_RESOURCES

API Call

HDSET_Open_Remote_Headset_Port(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam - 1)], TempParam->Params[1].intParam, FALSE, HDSET_Event_Callback, (unsigned long)0)

API Prototype

int BTPSAPI HDSET_Open_Remote_Headset_Port(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, unsigned int RemoteServerPort, Boolean_t SupportInBandRinging, HDSET_Event_Callback_t EventCallback, unsigned long CallbackParameter)

Description of API

The following function is responsible for Opening a Remote Headset Port on the specified Remote Device. This function accepts the Bluetooth Stack ID of the Bluetooth Stack which is to open the HDSET Connection as the first parameter. The second parameter specifies the Board Address (NON NULL) of the Remote Bluetooth Device to connect with. The next parameter specifies whether or not the Local Audio Gateway (the entity that is connecting to the Remote Headset) supports In Band Ringing or not (TRUE if supported). The final two parameters specify the HDSET Event Callback function, and callback parameter, respectively, of the HDSET Event Callback that is to process any further interaction with the specified Remote Port (Opening Status, Close Status, etc). This function returns a non-zero, positive, value if successful, or a negative return error code if this function is unsuccessful. If this function is successful, the return value will represent the HDSET Port ID that can be passed to all other functions that require it. Once a Remote Headset opened, it can only be closed via a call to the HDSET_Close_Port() function (passing the return value from this function).

CloseClient

Description

The following function is responsible for terminating a connection with a Remote Headset or Audio Gateway Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

It is not necessary to include parameters when using this command.

Possible Return Values

(0) HSP Server closed successfully

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000)

(-1001) BTHDSET_ERROR_NOT_INITIALIZED

(-1002) BTHDSET_ERROR_INVALID_BLUETOOTH_STACK_ID

API Call

HDSET_Close_Port(BluetoothStackID, HDSClientID

API Prototype

int BTPSAPI HDSET_Close_Port(unsigned int BluetoothStackID, unsigned int HDSETPortID)

Description of API

The following function exists to close a HDSET Port that was previously opened by any of the following mechanisms:

- Successful call to HDSET_Open_Remote_Headset_Port() function.
- Successful call to HDSET_Open_Remote_Audio_Gateway_Port() function.
- Incoming call request (Headset or Audio Gateway) which the Server was opened with either the HDSET_Open_Headset_Server_Port() or the HDSET_Open_Audio_Gateway_Server_Port() functions.

This function accepts as input the Bluetooth Stack ID of the Bluetooth Stack which the Open HDSET Port resides and the HDSET Port ID (return value from one of the above mentioned Open functions) of the Port to Close. This function returns zero if successful, or a negative return value if there was an error. This function does NOT Un-Register a HDSET Server Port from the system, it ONLY disconnects any connection that is currently active on the Server Port. The HDSET_Close_Server_Port() function can be used to Un-Register the HDSET Server Port. .

RingIndication

Description

The following function is responsible for sending a Ring Indication to the Remote connected Headset. This function returns zero if successful or a negative return error code if an error occurs.

Parameters

It is not necessary to include parameters when using this command.

Possible Return Values

(0) command sent successfully

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1001) BTHDSET_ERROR_NOT_INITIALIZED

(-1002) BTHDSET_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005) BTHDSET_ERROR_INVALID_OPERATION

API Call

HDSET_Ring_Indication(BluetoothStackID, ((ServerConnected)?HDSServerID:HDSClientID))

API Prototype

int BTPSAPI HDSET_Ring_Indication(unsigned int BluetoothStackID, unsigned int HDSETPortID)

Description of API

The following function is responsible for Sending a Ring Indication to the remote side. The function accepts the Bluetooth Stack ID of the Bluetooth Stack which has received the HDSET Connection Request and the HDSET Port ID for which the Connection has been established. This function returns a zero if successful, or a negative return error code if there was an error. .

ManageAudio

Description

The following function is responsible for setting up or releasing an audio connection. This function returns zero on successful execution and a negative value on all errors.

Parameters

The Manage Audio command requires only one parameter for the ManageAudio mode. This value must be specified as 0 (for Release) or 1 (for Setup).

Command Call Examples

"ManageAudio 0" Attempts to Release the Audio Connection from Server with port index 1.

"ManageAudio 1" Attempts to Setup the Audio Connection with Server port index 2.

Possible Return Values

(0) command sent successfully

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000) BTHDSET_ERROR_INVALID_PARAMETER

(-1001) BTHDSET_ERROR_NOT_INITIALIZED

(-1002) BTHDSET_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1005) BTHDSET_ERROR_INVALID_OPERATION

API Call

HDSET_Setup_Audio_Connection(BluetoothStackID, ((ServerConnected)?HDSServerID:HDSClientID), FALSE)

or HDSET_Release_Audio_Connection(BluetoothStackID, ((ServerConnected)?HDSServerID:HDSClientID))

API Prototype

int BTPSAPI HFRE_Setup_Audio_Connection(unsigned int BluetoothStackID, unsigned int HFREPortID)

or int BTPSAPI HDSET_Release_Audio_Connection(unsigned int BluetoothStackID, unsigned int HDSETPortID)

Description of API

This function is responsible for Setting Up an Audio Connection between the Local Audio Gateway and Remote Headset Device. This function may ONLY be used by an Audio Gateway. This function accepts as its input parameters the Bluetooth Stack ID for which the HDSET Port ID is valid as well as the HDSET Port ID (of the Audio Gateway). The final parameter specifies whether this is In-Band ringing (TRUE) or not (FALSE). If In-Band Ringing is specified then the remote Headset is required to accept the call. This function returns zero if successful or a negative return error code if there was an error.

(or) This function is responsible for Releasing an Audio Connection which was previously established by the local Audio Gateway or by a call to the HDSET_Setup_Audio_Connection() function. This function may ONLY be used by an Audio Gateway. This function accepts as its input parameters the Bluetooth Stack ID for which the HDSET Port ID is valid as well as the HDSET Port ID. This function returns zero if successful or a negative return error code if there was an error.

9 Map Demo Guide

9.1 Demo Overview

Note

The same instructions can be used to run this demo on either the Tiva, MSP432 or STM32F4 Platforms.

This demo allows users to evaluate TI's CC256x Bluetooth device by using the Tiva DK-TM4C129X, MSP432 or STM32F4. The SPP sample application code is provided to enable a rich out-of-box experience to the user. The application allows the user to use a console to send Bluetooth commands, setup a Bluetooth Device to accept connections, connect to a remote Bluetooth device and communicate over Bluetooth. The Message Access Profile (MAP) sample application code is provided to enable a rich out-of-box experience to the user. The application allows the user to use a console to send Bluetooth commands, setup a Bluetooth Device to accept connections, connect to a remote Bluetooth device and communicate over Bluetooth.

It is recommended that the user visits the kit setup Getting Started Guide for MSP-430 or Getting Started Guide for TIVA pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as **Tiva Virtual COM Port (COM x)**, **XDS110 Class Application/User UART (COM x)** for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud, 8, n, 1. Once connected, reset the device using Reset S3 button and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

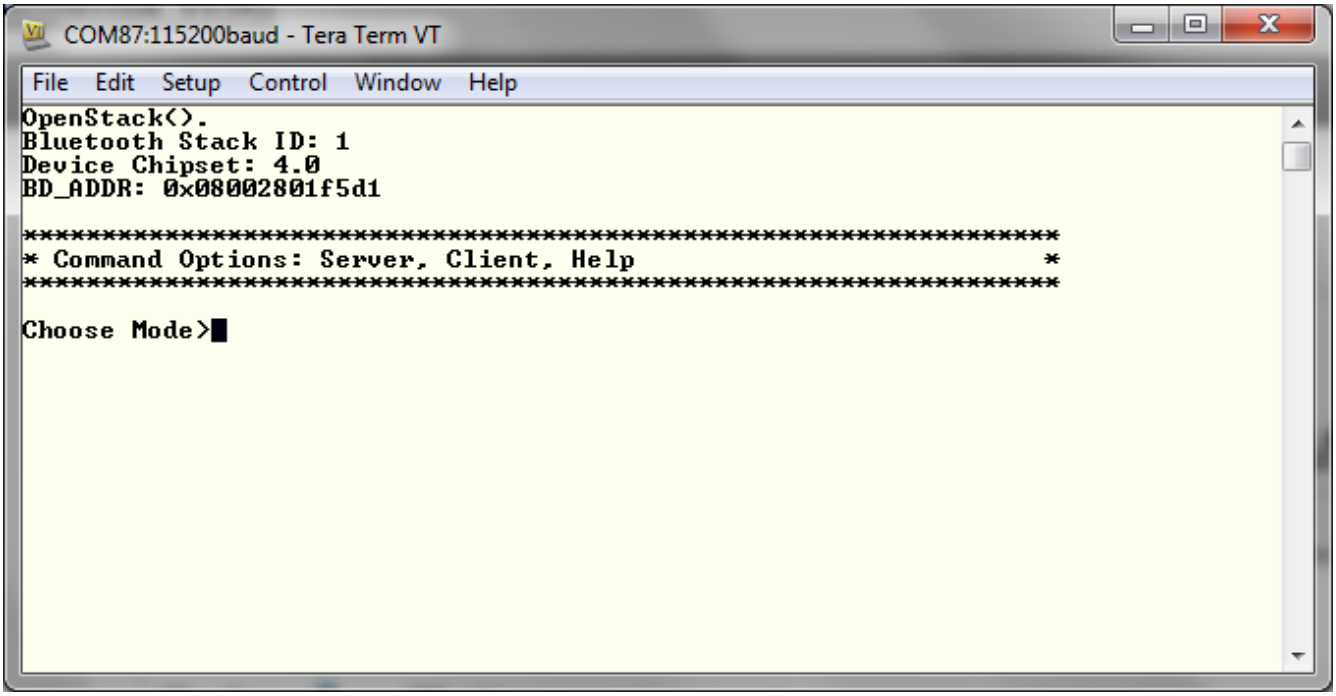

Figure 9-1. MAP Demo Initial Screen of PBAP Application

9.2 Demo Application

This section provides a description of how to use the demo application to connect two configured board and communicate over Bluetooth. Bluetooth MAP is a simple Client-Server connection process. We will setup one of the boards as a Server and the other board as a Client. We will then initiate a connection from the Client to the Server. Once connected, we can transmit data between the two devices over Bluetooth.

Server setup on the demo application

1. We will setup the first board as a Server. Perform the steps mentioned earlier in Running the Bluetooth Code section to initialize the application. Once initialized, note the Bluetooth address of the Server. We will later use this to initiate a connection from the Client.
2. On the Choose mode> prompt, enter Server.
3. You will see a list of all possible commands at this time for a Server. You can see this list at any time by issuing the Help command at the Server> prompt.
4. Now we are ready to open a Server. To open a Server, at the Server>prompt, issue the OpenServer 1 C:\Temp command. You can replace 1 with any number between 1 and 30, as long as there is no Server open on that port. The C:\Temp stands for the directory in which you store your received messages. Once you see the MAP_Open_Message_Access_Server() Successful notification, you have a MAP Server open on port 1.

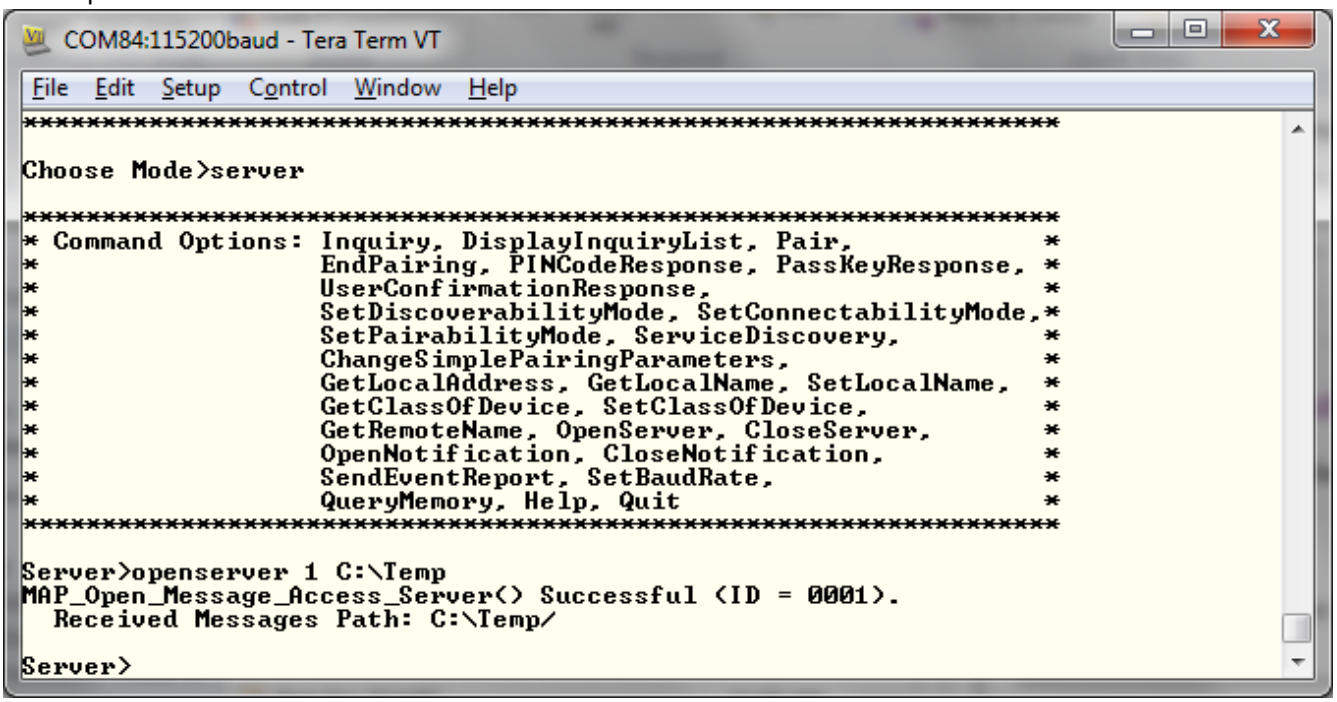

Figure 9-2. MAP Demo Server Setup

5. We will setup the second board as a Client. Perform the steps mentioned earlier in Running the Bluetooth Code section to initialize the application. On the Choose mode> prompt, enter Client.
6. You will see a list of all possible commands at this time for a Client. You can see this list at any time by issuing the Help command at the Client> prompt.
7. At the Client> prompt, issue the Inquiry command. This will initiate the Inquiry process. Once it is complete, you will get a list of all discovered devices.
8. You can access this list any time by issuing the DisplayInquiryList command at the Client prompt.

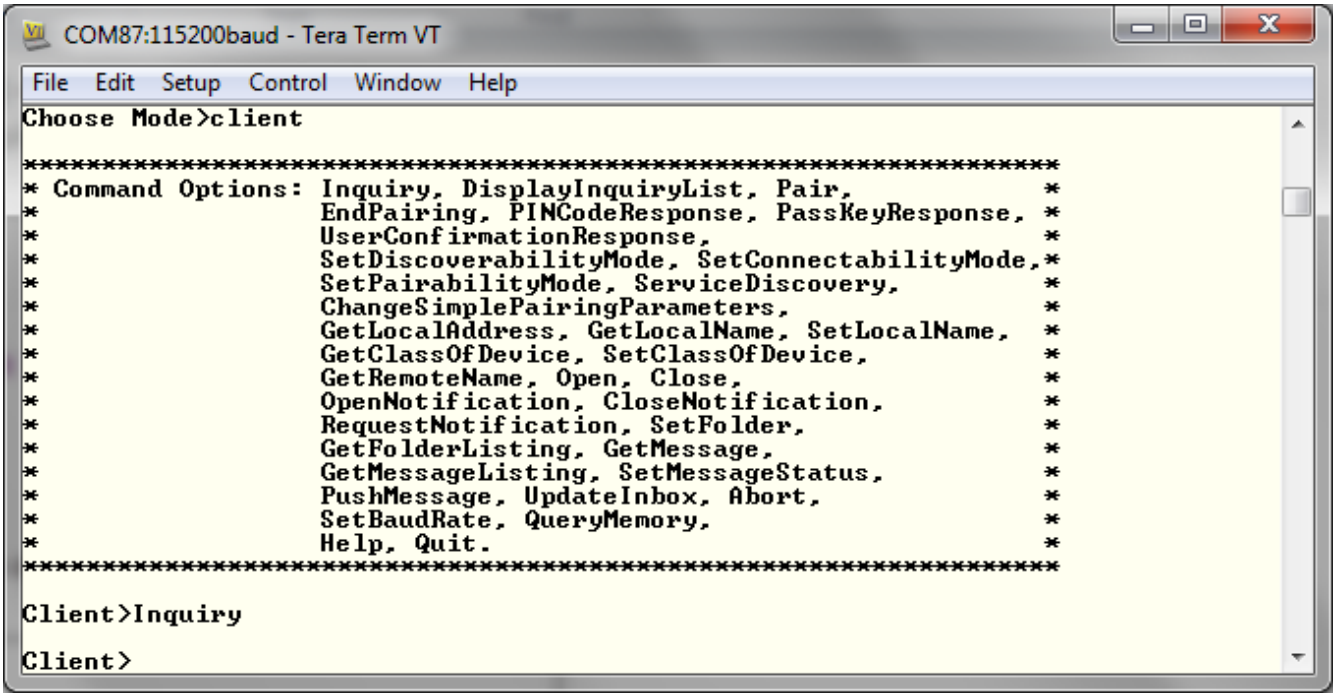

Figure 9-3. MAP Demo Client Setup

Initiating connection from the Client

9. Note the index number of the first board that was configured as a Server. [If the list is not on the screen, issue DisplayInquiryList command on the Client to display the list of discovered devices again.]
10. Issue a Open <index number> <Server port number> command at the command prompt.
11. Wait for the Open Port confirmation.
12. When a Client successfully connects to a Server, the Server will see the open port indication.

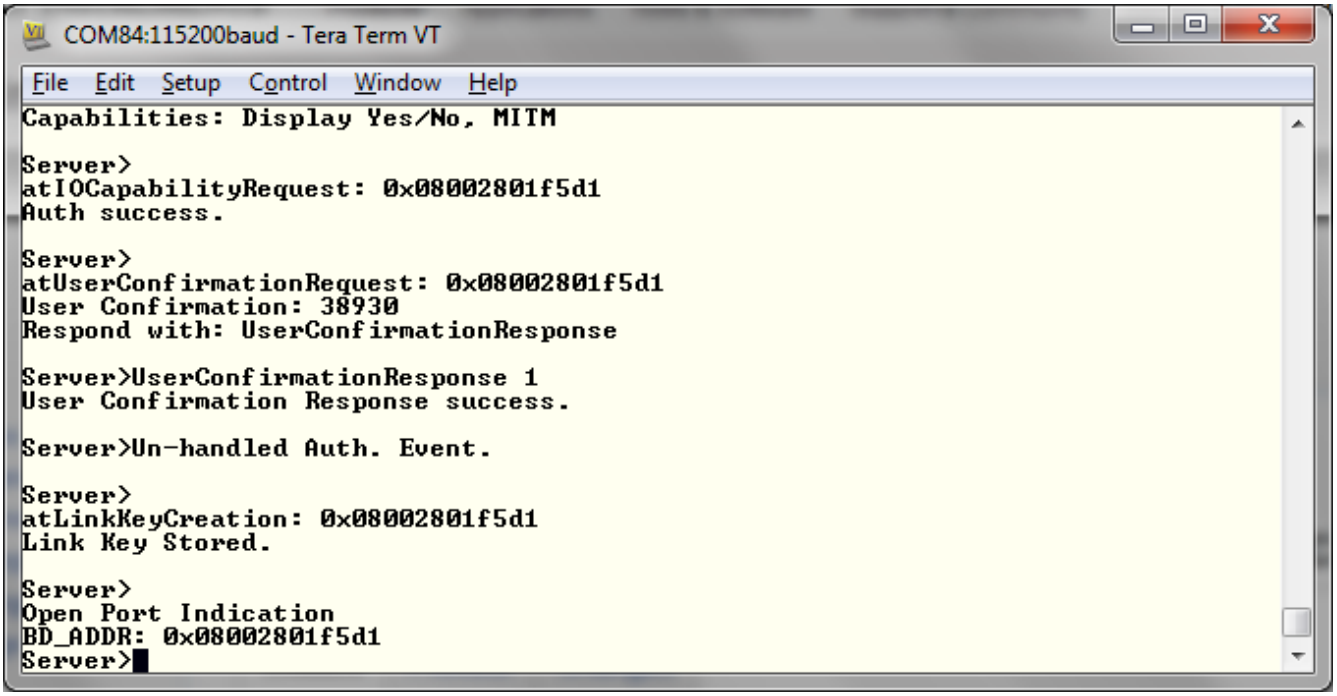

Figure 9-4. MAP Demo Server Connection

13. On the Client Side, open a notification Server issuing both the **OpenNotification 1** and **RequestNotification** commands . We should see on the **Server**, a Notification Registration Indication

and a MAP_Set_Notification_Registration_Response() Successful. On the **Client** Side you should see a Notification Registration Confirmation Result.

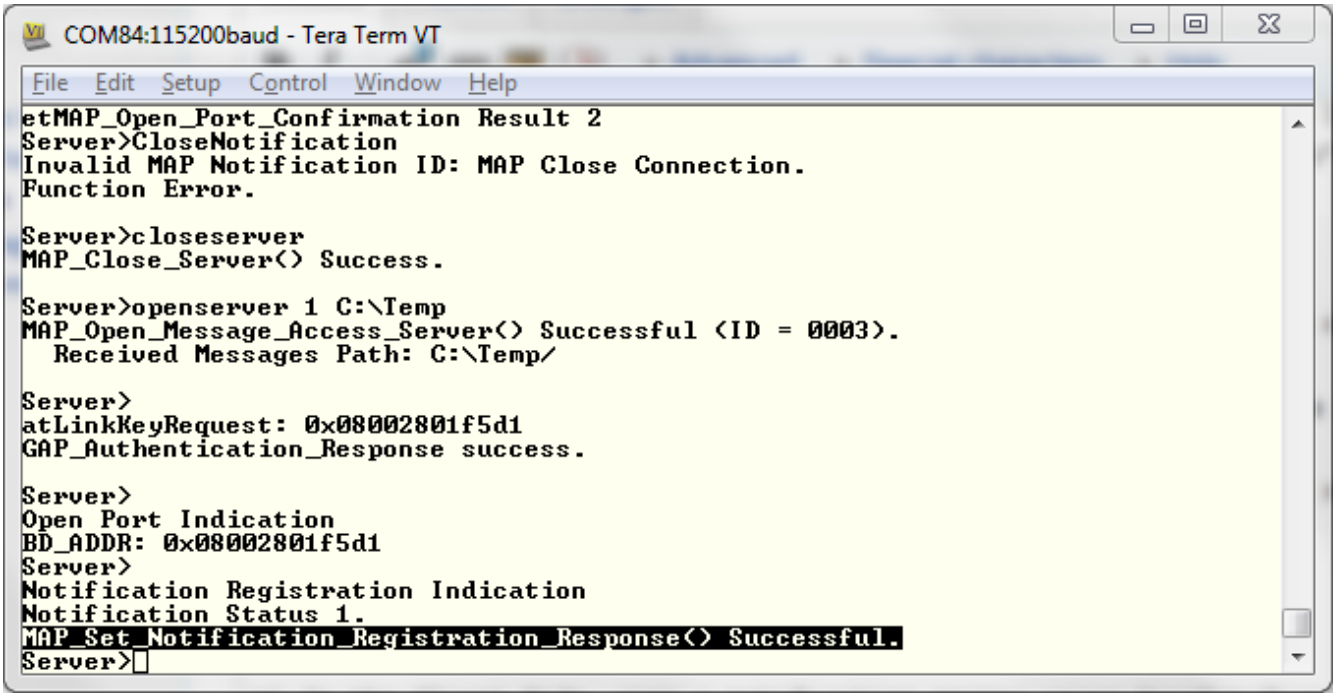

Figure 9-5. MAP Demo Client Connection 1

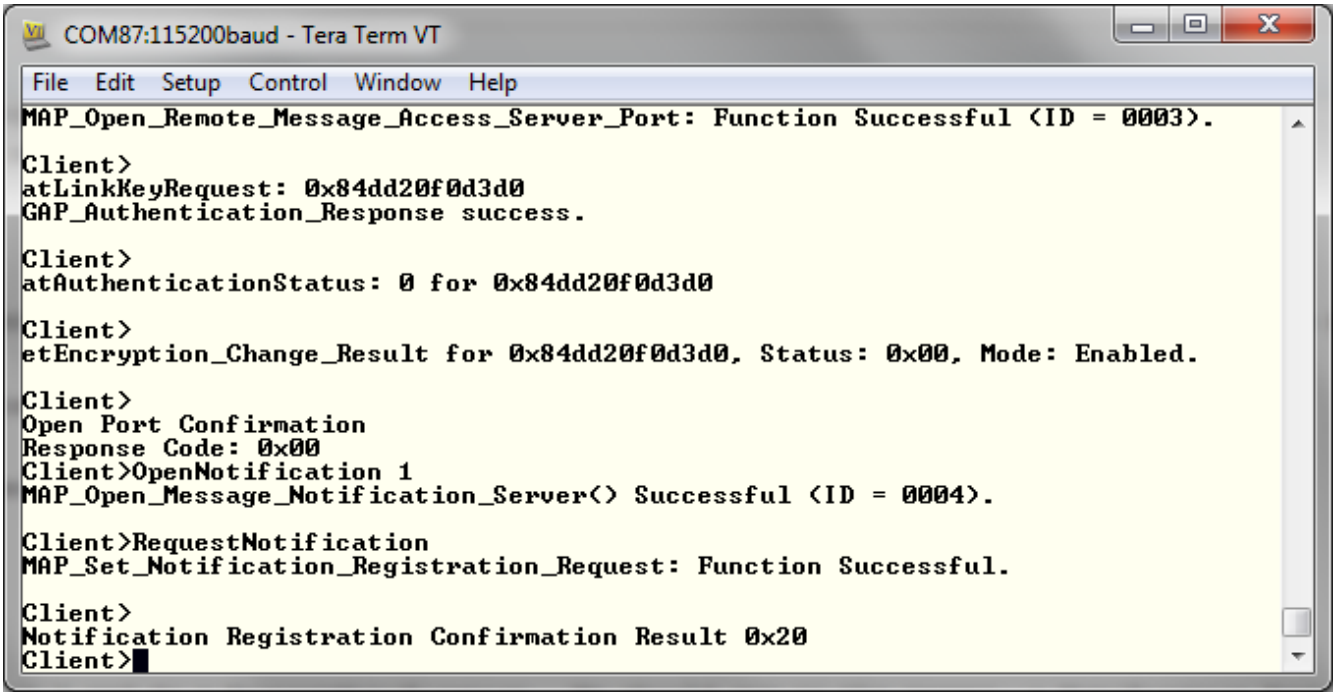

Figure 9-6. MAP Demo Client Connection 2

Data Transfer between Client and Server

14. Now we have a MAP connection established and both devices are ready to transmit data to each other. We can set and get folders. For example you can issue the **SetFolder 1** telecom command.
15. We can get a Message Listing and an individual message. For example you can issue the **GetMessageListing 65535 0** command.

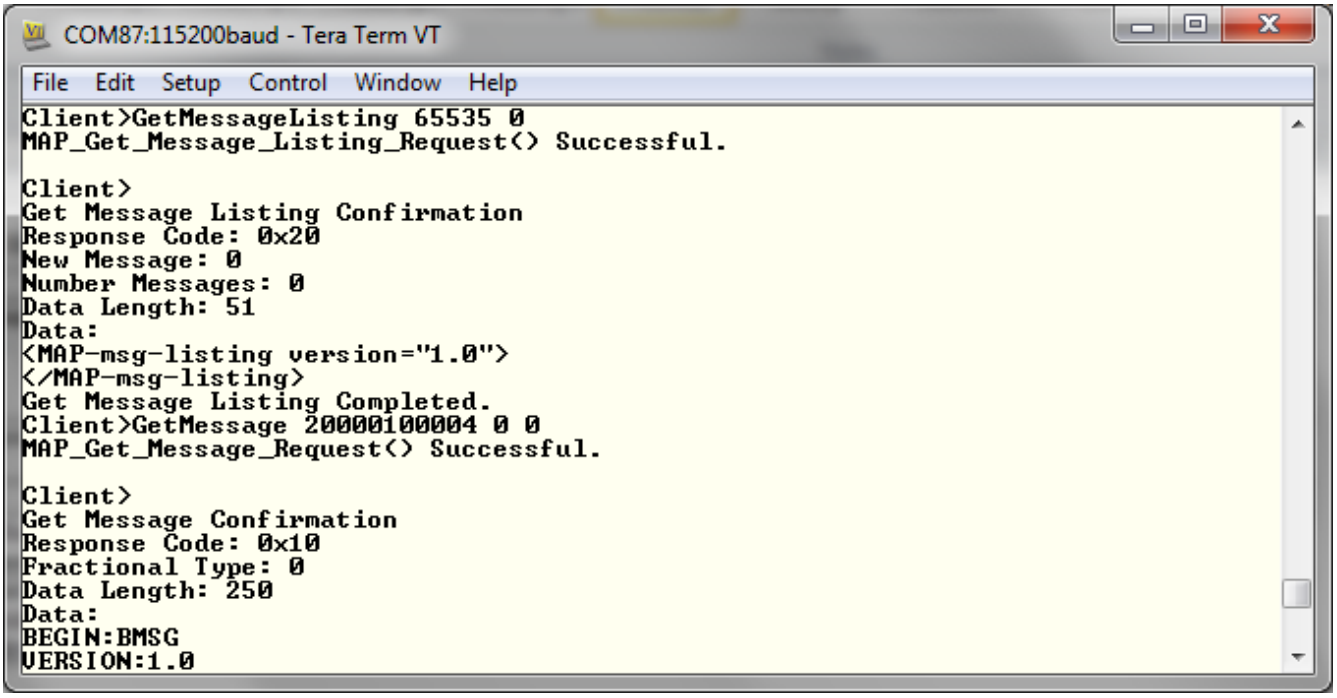

Figure 9-7. MAP Demo Client Connection 3

16. And we can set the message status, push messages and update the inbox. For example, you can issue the **PushMessage 1 1 20000100004 0 1** command.

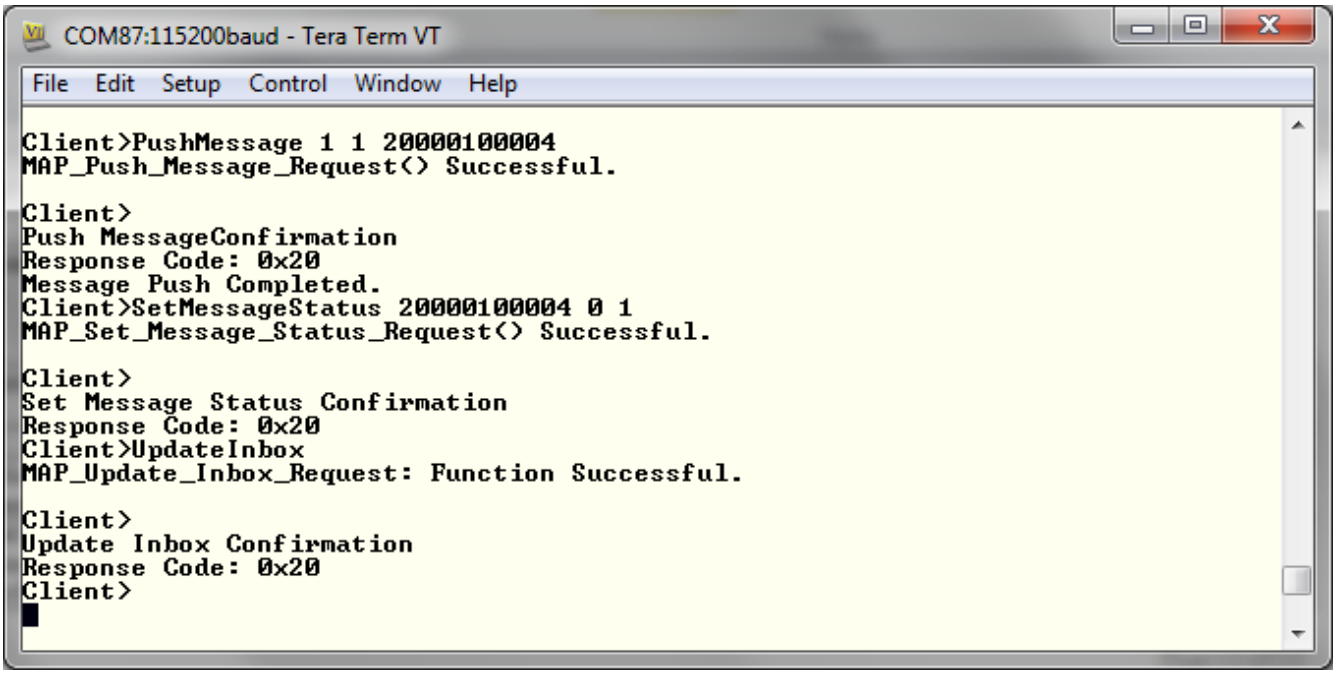

Figure 9-8. MAP Demo Client Connection 4

9.3 Application Command

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands

Message Access Profile Commands

OpenServer

Description

The following function is responsible for creating a local MAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

This command requires two parameters, the Port Number and the Received Messages Path.

Command Call Examples

Open 1 C:\Temp Attempts to Open a MAP Port Server at Port Number #1, with C:\Temp as the received messages path.

Possible Return Values

(0) MAP_Open_Message_Access_Server() Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Open_Message_Access_Server(BluetoothStackID, TempParam->Params[0].intParam, MAP_Event_Callback_Server, 0);

API Prototype

int BTPSAPI MAP_Open_Message_Access_Server(unsigned int BluetoothStackID, unsigned int MessageAccessServiceServerPort, MAP_Event_Callback_t EventCallback, unsigned long CallbackParameter)

Description of API

The following function is responsible for opening a local MAP Server. The first parameter is the Bluetooth Stack ID on which to open the Server. The second parameter is the Port on which to open this Server, and *MUST* be between MAP_PORT_NUMBER_MINIMUM and MAP_PORT_NUMBER_MAXIMUM. The third parameter is the Callback function to call when an event occurs on this Server Port. The final parameter is a user-defined callback parameter that will be passed to the callback function with each event. This function returns a positive, non zero value if successful or a negative return error code if an error occurs. A successful return code will be a MAP Profile ID that can be used to reference the Opened MAP Profile Server Port in ALL other MAP Server functions in this module. Once an MAP Profile Server is opened, it can only be Un-Registered via a call to the MAP_Close_Server() function (passing the return value from this function).

CloseServer

Description

The following function is responsible for deleting a local MAP Server that was created via a successful call to the OpenServer() function. This function returns zero if successful and a negative value if an error occurred.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Server closing

Possible Return Values

(0) MAP_Close_Server() Success

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Close_Server(BluetoothStackID, MAPID);

API Prototype

int BTPSAPI MAP_Close_Server(unsigned int BluetoothStackID, unsigned int MAPID)

Description of API

The following function is responsible for closing a currently open/registered Message Access Profile Server. This function is capable of closing servers opened via a call to MAP_Open_Message_Access_Server() and MAP_Open_Message_Notification_Server(). The first parameter to this function is the Bluetooth Stack ID of the Bluetooth Protocol Stack Instance that is associated with the Message Access Profile Server being closed. The second parameter to this function is the MAP ID of the Profile Server to be closed. This function returns zero if successful, or a negative return value if there was an error.

Note

This function only closes/un-registers servers it does NOT delete any SDP Service Record Handles that are registered for the specified Server.

Open

Description

The following function is responsible for initiating a connection with a Remote MAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

The command takes two parameters to work. The first is the Inquiry Index which can be found using the DisplayInquiryList command after an Inquiry has been completed. The second is the RFCOMM Server Port which will be used to open a Remote PBAP Port.

Command Call Examples

"Open 12 4" Attempts to Open a Remote Message Access Port Server with the Remote Bluetooth Device whose address is found at the twelfth Inquiry Index using RFCOMM Server Port #4." "Open 1 1" Attempts to Open a Remote Message Access Port Server with the Remote Bluetooth Device whose address is found at the first Inquiry Index using RFCOMM Server Port #1." "Open 19 3" Attempts to Open a Remote Message Access

Port Server with the Remote Bluetooth Device whose address is found at the nineteenth Inquiry Index using RFCOMM Server Port #3. "

Possible Return Values

(0)MAP_Open_Remote_Message_Access_Server_Port: Function Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Open_Remote_Message_Access_Server_Port(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam-1)], TempParam->Params[1].intParam, MAP_Event_Callback_Client, 0);

API Prototype

int BTPSAPI MAP_Open_Remote_Message_Access_Server_Port(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, unsigned int ServerPort, MAP_Event_Callback_t EventCallback, unsigned long CallbackParameter)

Description of API

The following function is responsible for opening a connection to a remote Message Access Server. The first parameter is the Bluetooth Stack ID of the local Bluetooth stack. The second parameter is the remote Bluetooth Device Address of the Bluetooth MAP Profile Server with which to connect. The third parameter specifies the remote Server port with which to connect. The final two parameters specify the MAP Profile Event Callback Function and the Callback Parameter to associate with this MAP Profile Client. The ServerPort parameter *MUST* be between MAP_PORT_NUMBER_MINIMUM and MAP_PORT_NUMBER_MAXIMUM. This function returns a positive, non zero, value if successful or a negative return error code if an error occurs. A successful return code will be a MAP ID that can be used to reference the remote opened MAP Profile Server in ALL other MAP Profile Client functions in this module. Once a remote Server is opened, it can only be closed via a call to the MAP_Close_Connection() function (passing the return value from this function).

Close

Description

The following function is responsible for terminating a connection with a remote MAP Client or Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Server closing.

Possible Return Values

(0) MAP_Close_Connection: Function Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Close_Connection(BluetoothStackID, MAPID);

API Prototype

int BTPSAPI MAP_Close_Connection(unsigned int BluetoothStackID, unsigned int MAPID)

Description of API

The following function is responsible for closing a currently ongoing MAP Profile connection. The first parameter is the Bluetooth Stack ID of the Bluetooth Protocol Stack Instance that is associated with the MAP Profile connection being closed. The second parameter to this function is the MAP ID of the MAP Profile connection to be closed. This function returns zero if successful, or a negative return value if there was an error.

Note

If this function is called with a Server MAP ID (a value returned from a call to MAP_Open_Server_Port()) any clients currently connected to this Server will be terminated, but the Server will remained open and registered. If this function is called using a Client MAP ID (a value returned from a call to MAP_Open_Remote_Server_Port()), the Client connection will be terminated/ closed entirely.

OpenNotification

Description

The following function is responsible for creating a local MAP Notification Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

The command only requires one parameter. This parameter is the Port Number.

Command Call Examples

"Open 1" Attempts to Open a MAP Notification Server at Port Number #1".

Possible Return Values

(0) MAP_Open_Message_Notification_Server() Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Open_Message_Notification_Server(BluetoothStackID, TempParam->Params[0].intParam, MAPID, MAP_Event_Callback_Client, 0);

API Prototype

int BTPSAPI MAP_Open_Message_Notification_Server(unsigned int BluetoothStackID, unsigned int MessageNotificationServiceServerPort, unsigned int MAS_MAPID, MAP_Event_Callback_t EventCallback, unsigned long CallbackParameter)

Description of API

The following function is responsible for opening a local MAP Server. The first parameter is the Bluetooth Stack ID on which to open the Server. The second parameter is the Port on which to open this Server, and *MUST* be between MAP_PORT_NUMBER_MINIMUM and MAP_PORT_NUMBER_MAXIMUM. The third parameter is the MAPID for the Message Access Client that the Server is being associated with. The fourth paramerer is a Callback function to call when an event occurs on this Server Port. The final parameter is a user-defined callback parameter that will be passed to the callback function with each event. This function returns a positive, non zero value if successful or a negative return error code if an error occurs. A successful return code will be a MAP Profile ID that can be used to reference the Opened MAP Profile Server Port in ALL other MAP Server functions in this module. Once an MAP Profile Server is opened, it can only be Un-Registered via a call to the MAP_Close_Server() function (passing the return value from this function)

CloseNotification

Description

The following function is responsible for deleting a local MAP Notification Server that was created via a successful call to the OpenNotificationServer() function. This function returns zero if successful and a negative value if an error occurred.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Server closing.

Possible Return Values

(0) MAP_Close_Server() Success

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Close_Server(BluetoothStackID, MAPNID);

API Prototype

int BTPSAPI MAP_Close_Server(unsigned int BluetoothStackID, unsigned int MAPID)

Description of API

The following function is responsible for closing a currently open/registered Message Access Profile Server. This function is capable of closing servers opened via a call to MAP_Open_Message_Access_Server() and MAP_Open_Message_Notification_Server(). The first parameter to this function is the Bluetooth Stack ID of the Bluetooth Protocol Stack Instance that is associated with the Message Access Profile Server being closed. The second parameter to this function is the MAP ID of the Profile Server to be closed. This function returns zero if successful, or a negative return value if there was an error.

Note

This function only closes/un-registers servers it does NOT delete any SDP Service Record Handles that are registered for the specified Server.

RequestNotification

Description

The following function is responsible for issuing a MAP Request Notification Request to the remote MAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

The command only requires one parameter, which is the connection index, 0 = Disconnect, 1 = Connect.

Command Call Examples

RequestNotification 1 makes a MAP Request Notification connection to the remote MAP Server.
RequestNotification 0 makes a MAP Request Notification disconnection to the remote MAP Server.

Possible Return Values

(0) MAP_Set_Notification_Registration_Request: Function Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Set_Notification_Registration_Request(BluetoothStackID, MAPID, (Boolean_t)((TempParam->Params[0].intParam)?TRUE:FALSE));

API Prototype

int BTPSAPI MAP_Set_Notification_Registration_Request(unsigned int BluetoothStackID, unsigned int MAPID, Boolean_t Enabled)

Description of API

The following function is responsible for providing a mechanism to Enable or Disable Notification messages from the remote Message Access Server. The first parameter to this function is the Bluetooth Stack ID. The second parameter to this function is the MAP ID of the Service Client making this call. The third parameter specifies if the Notifications should be Enabled or Disabled. This function returns zero if successful, or a negative return value if there was an error.

SetFolder

Description

The following function is responsible for issuing a MAP Set Folder Request to a remote MAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

This functions requires one parameter (path option parameter 0-Root, 1-Down, 2-Up). If down is selected as the path option then we require an additional paramters which is the path.

Command Call Examples

SetFolder 1 telecom uses down as the path option and telecom as the folder.

Possible Return Values

(0) MAP_Set_Folder_Request() Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Set_Folder_Request(BluetoothStackID, MAPID, (MAP_Set_Folder_Option_t)TempParam->Params[0].intParam, TempBuffer);

API Prototype

*int BTPSAPI MAP_Set_Folder_Request(unsigned int BluetoothStackID, unsigned int MAPID, MAP_Set_Folder_Option_t PathOption, Word_t *FolderName)*

Description of API

The following function generates a MAP Set Folder Request to the specified remote MAP Server.The BluetoothStackID parameter the ID of the Bluetooth Stack that is associated with this MAP Client. The MAPID parameter specifies the MAP ID for the local MAP Client. The PathOption parameter contains an enumerated value that indicates the type of path change to request. The FolderName parameter contains the folder name to include with this SetFolder request. This value can be NULL if no name is required for the selected PathOption.

See the MAP specification for more information. This function returns zero if successful or a negative return error code if there was an error.

Note

A successful return code does not mean that the remote MAP Profile Server successfully processed the command. The caller needs to check the confirmation result to determine if the remote MAP Profile Server successfully executed the Request. There can only be one outstanding MAP Profile Request active at any one time. Because of this, another MAP Profile Request cannot be issued until either the current request is Aborted (by calling the MAP_Abort_Request() function) or the current Request is completed (this is signified by receiving a Confirmation Event in the MAP Profile Event Callback that was registered when the MAP Profile Port was opened).

GetFolderListing

Description

The following function is responsible for issuing a MAP Get Folder Listing Request to a remote MAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

The command requires 2 parameters, the Max List Count, the List Start Offset.

Command Call Examples

GetFolderListing 65535 0 has 65535 as the Max List Count, 0 as the List Start Offset.

Possible Return Values

(0) MAP_Get_Folder_Listing_Request() Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Get_Folder_Listing_Request(BluetoothStackID, MAPID, (Word_t)TempParam->Params[0].intParam, (Word_t)TempParam->Params[1].intParam)

API Prototype

int BTPSAPI MAP_Get_Folder_Listing_Request(unsigned int BluetoothStackID, unsigned int MAPID, Word_t MaxListCount, Word_t ListStartOffset)

Description of API

The following function generates a MAP Get Folder Listing Request to the specified remote MAP Server. The BluetoothStackID parameter the ID of the Bluetooth Stack that is associated with this MAP Client. The MAPID parameter specifies the MAP ID for the local MAP Client (returned from a successful call to the MAP_Connect_Remote_Server_Port() function). The MaxListCount parameter is an unsigned integer that specifies the maximum number of list entries the Client can handle. A MaxListCount of ZERO (0) indicates that

this is a request for the number accessible folders in the current folder. The ListStartOffset parameter specifies the index requested by the Client in this Folder Listing. This function returns zero if successful or a negative return error code if there was an error.

Note

A successful return code does not mean that the remote MAP Profile Server successfully processed the command. The caller needs to check the confirmation result to determine if the remote MAP Profile Server successfully executed the Request. There can only be one outstanding MAP Profile Request active at any one time. Because of this, another MAP Profile Request cannot be issued until either the current request is Aborted (by calling the MAP_Abort_Request() function) or the current Request is completed (this is signified by receiving a Confirmation Event in the MAP Profile Event Callback that was registered when the MAP Profile Port was opened).

GetMessage

Description

The following function is responsible for issuing a MAP Get Message Request to a remote MAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

The command requires 3 parameters Message Handle, Char Set Index(0 = Native, 1 = UTF8), Fractional Type Index(0 = Unfragmented, 1 = First, 4 = Last).

Command Call Examples

GetMessage 20000100004 0 0 has 20000100004 as Message Handle, 0 as Native Char Set, 0 as Fractional Type Index.

Possible Return Values

(0) MAP_Get_Message_Request() Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Get_Message_Request(BluetoothStackID, MAPID, TempParam->Params[0].strParam, FALSE, CharSet, (MAP_Fractional_Type_t)TempParam->Params[1].intParam);

API Prototype

*int BTPSAPI MAP_Get_Message_Request(unsigned int BluetoothStackID, unsigned int MAPID, char *MessageHandle, Boolean_t Attachment, MAP_CharSet_t CharSet, MAP_Fractional_Type_t FractionalType)*

Description of API

The following function generates a MAP Get Message Request to the specified remote MAP Server. The BluetoothStackID parameter the ID of the Bluetooth Stack that is associated with this MAP Client. The MAPID

parameter specifies the MAP ID for the local MAP Client. The MessageHandle parameter is a pointer to a 16 byte Null terminated Unicode Text string that identifies the message. The Attachment parameter is used to indicate if any existing attachments to the specified message are to be included in the response. The CharSet and Fractional Type parameters specify that format of the response message. This function returns zero if successful or a negative return error code if there was an error.

Note

Specifying the FractionalType as ftUnfragmented causes no FractionalType Header to be added to the OBEX Header List. This is the value that should be specified for a message that is non-fragmented.

GetMessageListing

Description

The following function is responsible for issuing a MAP Get Message Listing Request to a remote MAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

The command requires 2 parameters, the Max List Count and the List Start Offset

Command Call Examples

GetMessageListing 65535 0 has 65535 as the Max List Count, 0 as the List Start Offset.

Possible Return Values

API Call

MAP_Get_Message_Listing_Request(BluetoothStackID, MAPID, Folder?TempBuffer:NULL, (Word_t)TempParam->Params[0].intParam, (Word_t)TempParam->Params[1].intParam, NULL);

API Prototype

*int BTPSAPI MAP_Get_Message_Listing_Request(unsigned int BluetoothStackID, unsigned int MAPID, Word_t *FolderName, Word_t MaxListCount, Word_t ListStartOffset, MAP_Message_Listing_Info_t *ListingInfo)*

Description of API

The following function generates a MAP Get Message Listing Request to the specified remote MAP Server. The BluetoothStackID parameter the ID of the Bluetooth Stack that is associated with this MAP Client. The MAPID parameter specifies the MAP ID for the local MAP Client (returned from a successful call to the MAP_Connect_Remote_Server_Port() function). The FolderName specifies the Folder from which the Message Listing is to be retrieved. If the parameter is NULL, the listing is made from the current directory. The MaxListCount parameter is an unsigned integer that specifies the maximum number of list entries the Client can handle. A MaxListCount of ZERO (0) indicates that this is a request for the number of messages in the specified folder. The ListStartOffset parameter specifies the index requested by the Client in this Listing. The ListInfo structure is used to specify a number of filters and options that should be applied to the request. This function returns zero if successful or a negative return error code if there was an error.

Note

A successful return code does not mean that the remote MAP Profile Server successfully processed the command. The caller needs to check the confirmation result to determine if the remote MAP Profile Server successfully executed the Request.

SetMessageStatus

Description

The following function is responsible for issuing a MAP Set Message Status Request to the remote MAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

The command requires 3 parameters Message Handle, Status Indicator Index(0 = Native, 1 = UTF8), Status Value Index(0 = Unfragmented, 1 = First, 4 = Last).

Command Call Examples

SetMessageStatus 20000100004 0 0 has 20000100004 as Message Handle, 0 as Status Indicator, 0 as Status Value.

Possible Return Values

(0) MAP_Set_Message_Status_Request() Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Set_Message_Status_Request(BluetoothStackID, MAPID, TempParam->Params[0].strParam, (MAP_Status_Indicator_t)TempParam->Params[1].intParam, TempParam->Params[2].intParam)

API Prototype

*int BTPSAPI MAP_Set_Message_Status_Request(unsigned int BluetoothStackID, unsigned int MAPID, char *MessageHandle, MAP_Status_Indicator_t StatusIndicator, Boolean_t StatusValue)*

Description of API

The following function generates a MAP Set Message Status Request to the specified remote MAP Server. The BluetoothStackID parameter the ID of the Bluetooth Stack that is associated with this MAP Client. The MAPID parameter specifies the MAP ID for the local MAP Client. The MessageHandle parameter is a pointer to a 16 byte Null terminated Unicode Text string that identifies the message. The StatusIndicator identifies the Status indicator to set. The StatusValue indicates the new state of the indicator. This function returns zero if successful or a negative return error code if there was an error.

PushMessage

Description

The following function is responsible for issuing a MAP Push Message Request to a remote MAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

This command requires 3 parameters, Transparent Index (0 = False, 1 = True), Retry Index (0 = False, 1 = True) and Message File Name.

Command Call Examples

PushMessage 0 0 20000100004 has as , 0 for Transparent as , 0 for Retry and 20000100004 as Message Handle.

Possible Return Values

(0) MAP_Push_Message_Request() Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

*MAP_Push_Message_Request(BluetoothStackID, MAPID, NULL, TempParam->Params[0].intParam, TempParam->Params[1].intParam, csUTF8, CurrentBufferSize, (Byte_t *)CurrentBuffer, &CurrentBufferSent, TRUE);*

API Prototype

*int BTPSAPI MAP_Push_Message_Request(unsigned int BluetoothStackID, unsigned int MAPID, Word_t *FolderName, Boolean_t Transparent, Boolean_t Retry, MAP_CharSet_t CharSet, unsigned int DataLength, Byte_t *DataBuffer, unsigned int *AmountSent, Boolean_t Final)*

Description of API

The following function generates a MAP Push Message Request to the specified remote MAP Server. The BluetoothStackID parameter the ID of the Bluetooth Stack that is associated with this MAP Client. The MAPID parameter specifies the MAP ID for the local MAP Client. The FolderName parameter specifies the destination location for the message. If this parameter is NULL, the current directory is used. The Transparent parameter is used to indicate that no copy of the message should be placed in the Sent Folder. Retry parameter is used to indicate if any attempts to retry the send if the previous attempt fails. The CharSet parameters specify that format of the message. The DataLength parameter defines the number of bytes that are included in the object segment (DataBuffer). The DataBuffer parameter is a pointer to a byte buffer containing the Message Listing Object segment. The AmountSent parameter is a pointer to variable which will be written with the actual amount of data that was able to be included in the packet. The final parameter to this function is a Boolean Flag indicating if this is the last segment of the object. This function returns zero if successful or a negative return error code if there was an error.

Note

The MessageObject is a "x-bt/message" character stream that is formatted as defined in the Message Access Profile Specification. Including a DataBuffer pointer and setting DataLength > 0 will cause a Body or End-of-Body header to be added to the packet, either on the first or subsequent packets. If the stack cannot include all the requested object (DataLength) in the current packet, a Body header will be used and AmountSent will reflect that not all of the data was sent. If all data is included, an End-of-Body header will be used. If AmountSent returns an amount smaller than the specified DataLength, not all the data was able to be sent. This function should be called again with an adjusted DataLength and DataBuffer pointer to account for the data that was successfully sent.

UpdateInbox

Description

The following function is responsible for issuing a MAP UpdateInbox Request to the remote MAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Server closing.

Possible Return Values

(0) MAP_Update_Inbox_Request: Function Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Update_Inbox_Request(BluetoothStackID, MAPID)

API Prototype

int BTPSAPI MAP_Update_Inbox_Request(unsigned int BluetoothStackID, unsigned int MAPID)

Description of API

The following function generates a MAP Update Inbox Request to the specified remote MAP Server. The BluetoothStackID parameter the ID of the Bluetooth Stack that is associated with this MAP Client. The MAPID parameter specifies the MAP ID for the local MAP Client. This function returns zero if successful or a negative return error code if there was an error.

Abort

Description

The following function is responsible for issuing an Abort with a remote MAP Server by issuing an OBEX Abort Command. This function returns zero if successful and a negative value if an error occurred.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Server closing.

Possible Return Values

(0) MAP_Abort_Request: Function Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTMAP_ERROR_INVALID_PARAMETER

(-1001) BTMAP_ERROR_NOT_INITIALIZED

(-1002) BTMAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTMAP_ERROR_DLL_INITIALIZATION_ERROR br> (-1004) BTMAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTMAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTMAP_ERROR_ACTION_NOT_ALLOWED

(-1007) BTMAP_ERROR_INSUFFICIENT_PACKET_LENGTH

API Call

MAP_Abort_Request(BluetoothStackID, MAPID);

API Prototype

int BTPSAPI MAP_Abort_Request(unsigned int BluetoothStackID, unsigned int MAPID)

Description of API

The following function is responsible for sending an Abort Request to the remote Server. The BluetoothStackID parameter the ID of the Bluetooth Stack that is associated with this MAP Client. The MAPID parameter specifies the MAP ID for the local MAP Client. This function returns zero if successful, or a negative return value if there was an error.

Note

Upon the reception of the Abort Confirmation Event it may be assumed that the currently on going transaction has been successfully aborted and new requests may be submitted.

10 PBAP Demo Guide

10.1 Demo Overview

Note

The same instructions can be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms.

This demo allows users to evaluate TI's CC256x Bluetooth device by using the Tiva DK-TM4C129X, MSP432 or STM32F4. The **Phone Book Access Profile (PBAP)** sample application code is provided to enable a rich out-of-box experience to the user. The application allows the user to use a console to send Bluetooth commands, setup a Bluetooth Device to accept connections, connect to a remote Bluetooth device and communicate over Bluetooth.

It is recommended that the user visits the kit setup Getting Started Guide for TIVA, Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as **Tiva Virtual COM Port (COM x)**, **XDS110 Class Application/User UART (COM x)** for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud, 8, n, 1. Once connected, reset the device using Reset S3 button and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

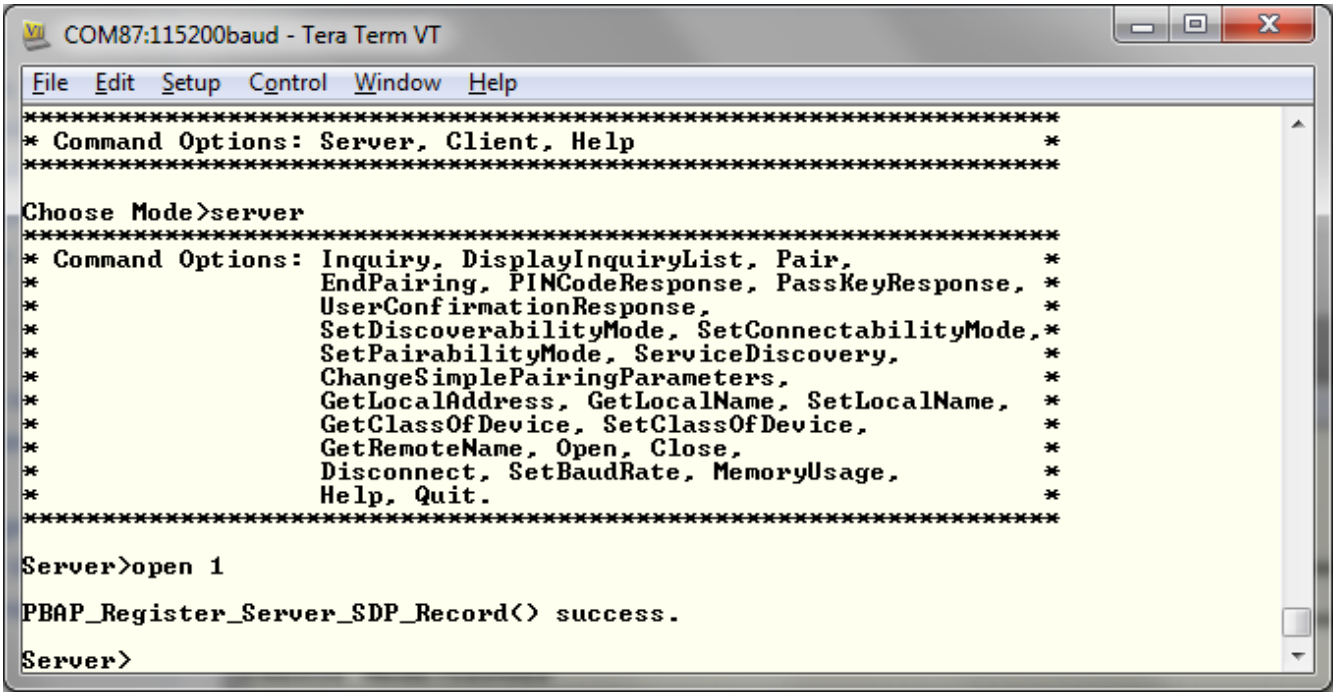

Figure 10-1. PBAP Demo Initial Screen of PBAP Application

10.2 Demo Application

This section provides a description of how to use the demo application to connect two configured board and communicate over Bluetooth. Bluetooth PBAP is a simple Client-Server connection process. We will setup one of the boards as a Server and the other board as a Client. We will then initiate a connection from the Client to the Server. Once connected, we can transmit data between the two devices over Bluetooth.

Server setup on the demo application

1. We will setup the first board as a Server. Perform the steps mentioned earlier in **Running the Bluetooth Code** section to initialize the application. Once initialized, note the Bluetooth address of the Server. We will later use this to initiate a connection from the Client.
2. On the **Choose mode>** prompt, issue the Server command.
3. You will see a list of all possible commands at this time for a Server. You can see this list at any time by entering **Help** at the **Server>** prompt.
4. Now we are ready to open a Server. To open a Server, at the **Server>** prompt, issue the **Open 1**. You can replace 1 with any number between 1 and 30, as long as there is no Server open on that port. Once you see **PBAP_Register_Server_SDP_Record() success**, you have a PBAP Server open on port 1.

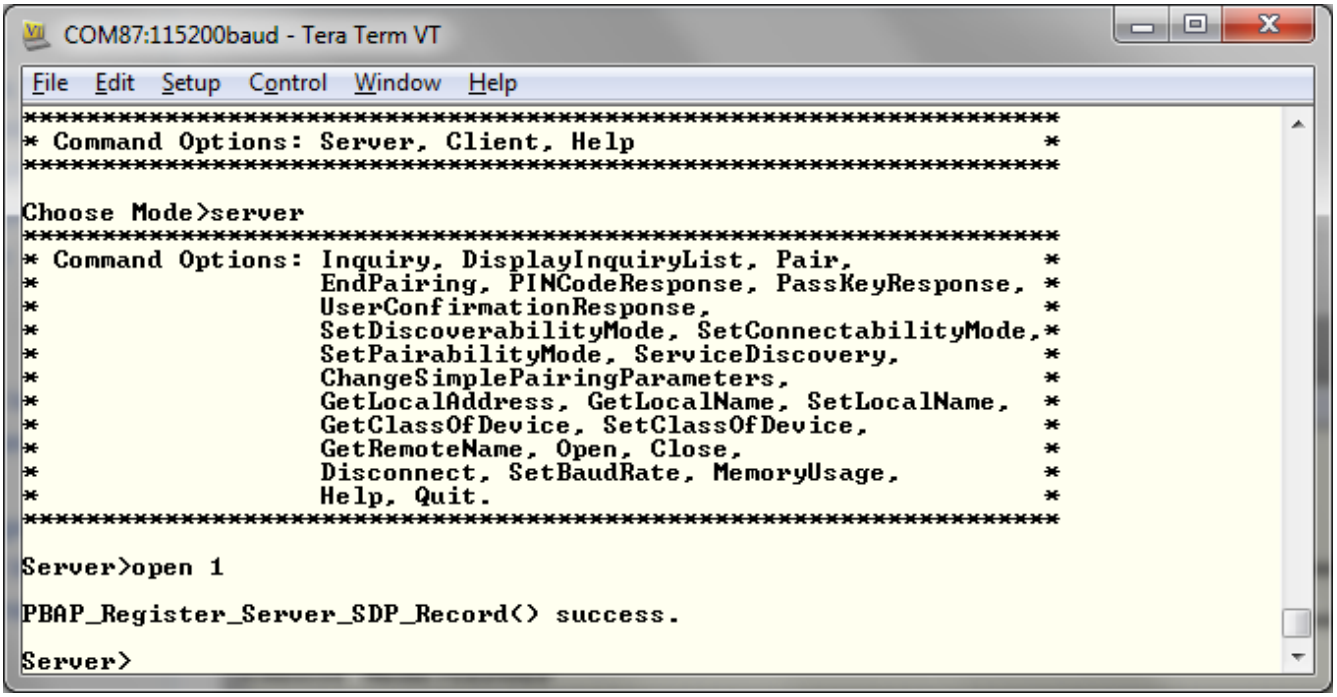

Figure 10-2. PBAP Demo Server Setup

Client setup and device discovery on the demo application

1. We will setup the second board as a Client. Perform the steps mentioned earlier in "Running the Bluetooth Code" section to initialize the application. On the **Choose mode>** prompt, enter **Client**.
2. You will see a list of all possible commands at this time for a Client. You can see this list at any time by issuing the **Help** at the **Client>** prompt.
3. At the **Client>** prompt, issue the Inquiry command. This will initiate the **Inquiry** process. Once it is complete, you will get a list of all discovered devices.
4. You can access this list any time by issuing the **DisplayInquiryList** at the Client prompt.

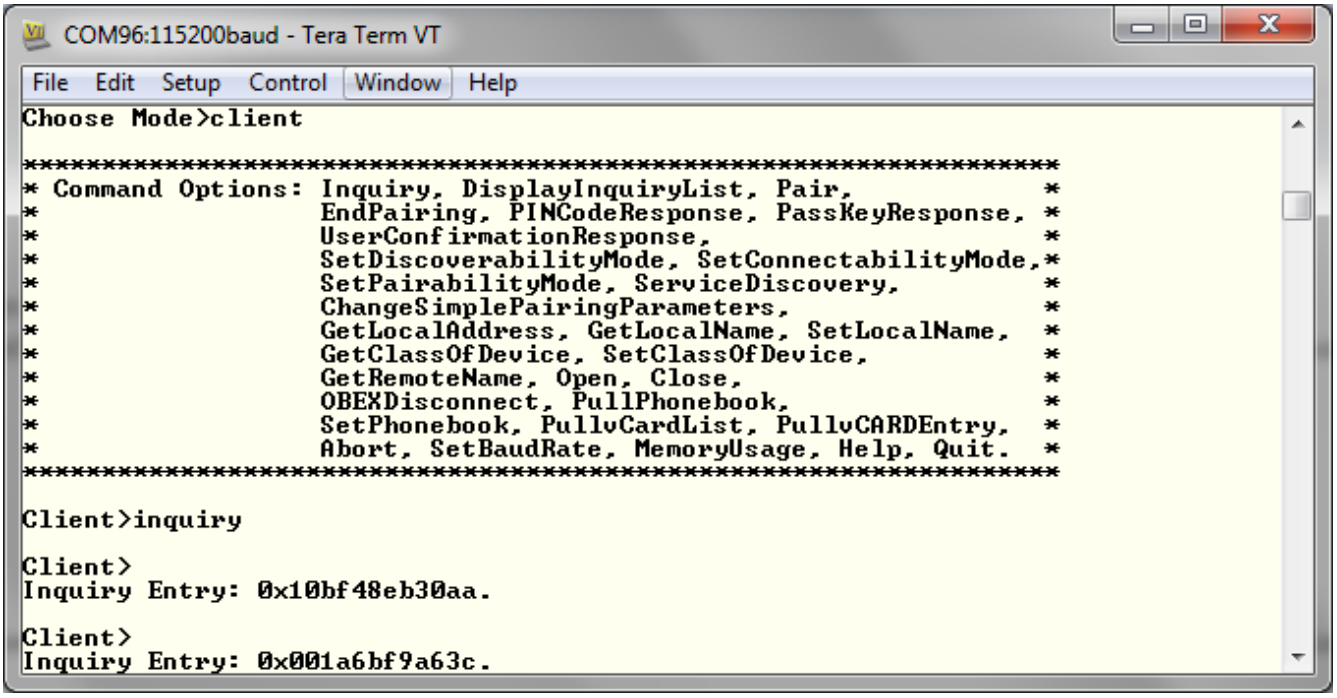

Figure 10-3. PBAP Client Setup

Initiating connection from the Client

1. Note the index number of the first board that was configured as a Server. [If the list is not on the screen, issue **DisplayInquiryList** command on the Client to display the list of discovered devices again.]
2. Issue a **Open <index number> <Server port number>** command at the command prompt.

> **Note**
>
> Port 19 must be used when connecting to Android phones and port 13 for IPhones.

3. Wait for the PBAP Open confirmation.
4. When a Client successfully connects to a **Server**, the Server will see the open indication.

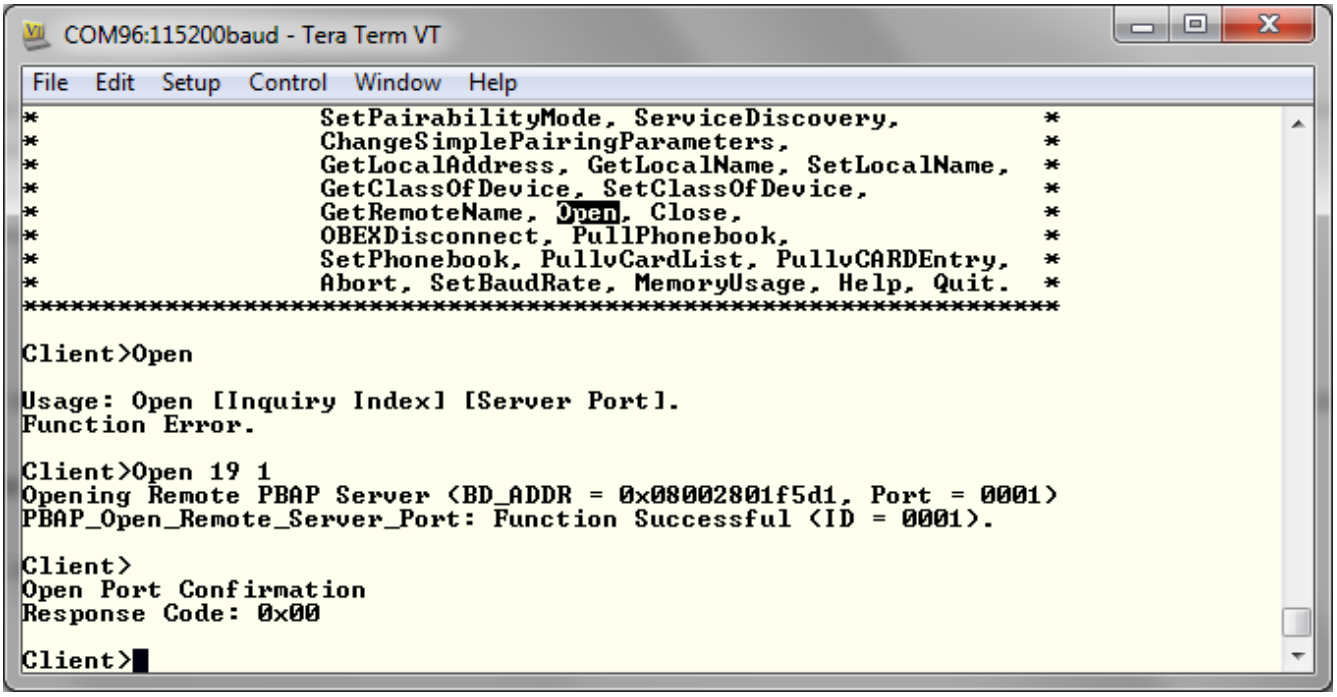

Figure 10-4. PBAP Demo Client Connection

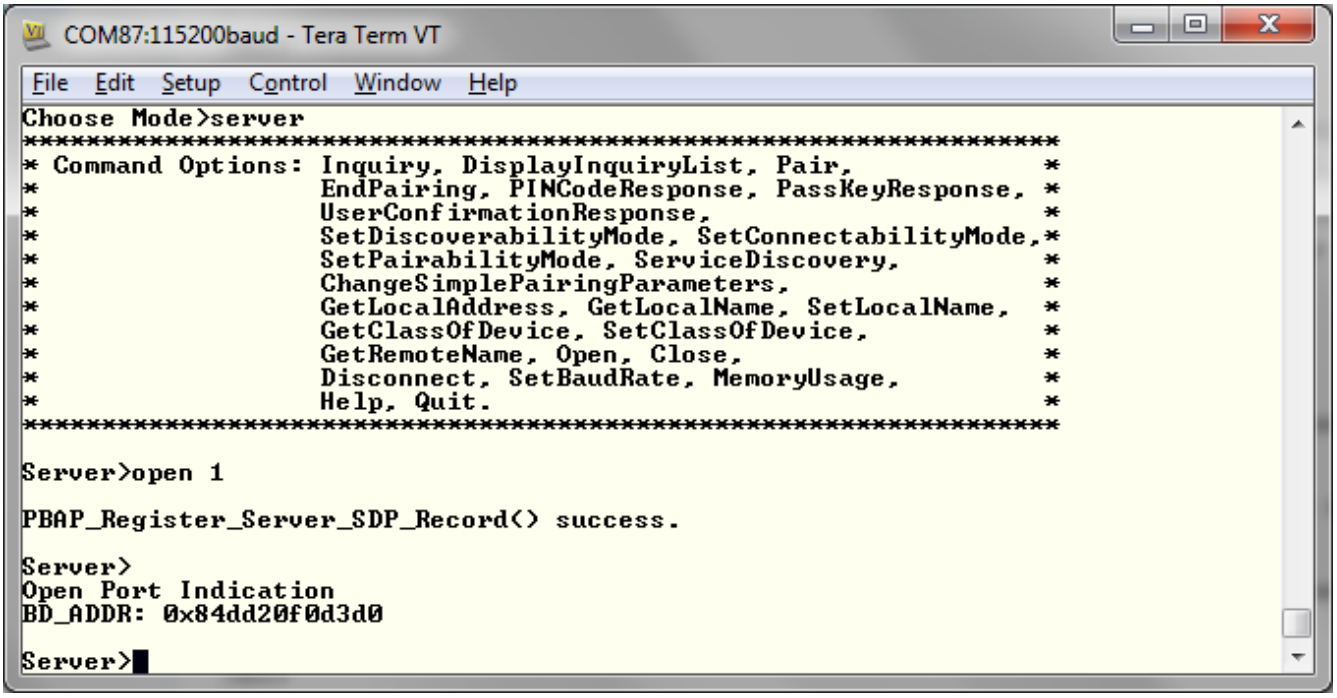

Figure 10-5. PBAP Demo Server Connection

Data Transfer between Client and Server

1. Now we have a PBAP connection established and both devices are ready to transmit data to each other.
2. We can Set Directories and Pull Phonebooks from Client by issuing the **SetPhonebook <Path Change> <Path Name>** and PullPhonebook **<Max List Count> <List Start Offset> <Phonebook Path>** commands.

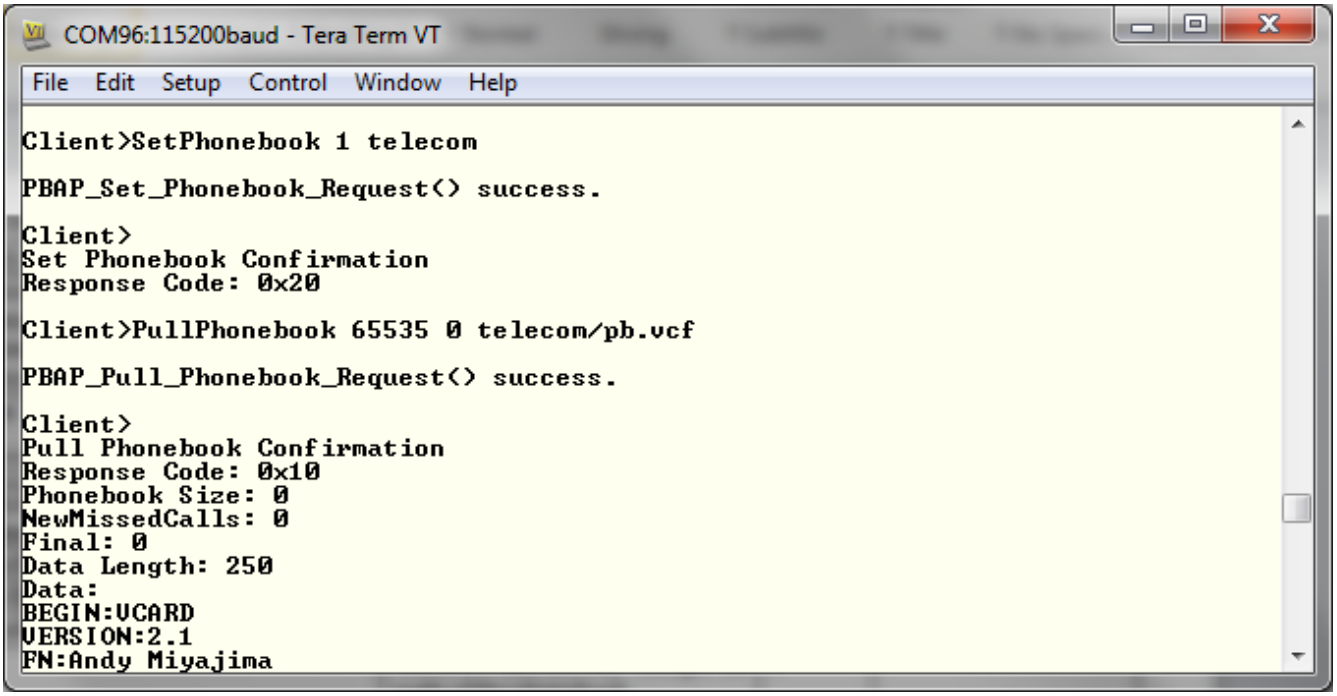

Figure 10-6. PBAP Demo Client Connection 2

3. We can also get vCardListings and individual vCard Entries by issuing the PullvCardList **<Max List Count> <List Start Offset> <Phonebook Path>** and **PullvCardEntry <vCard Name>**

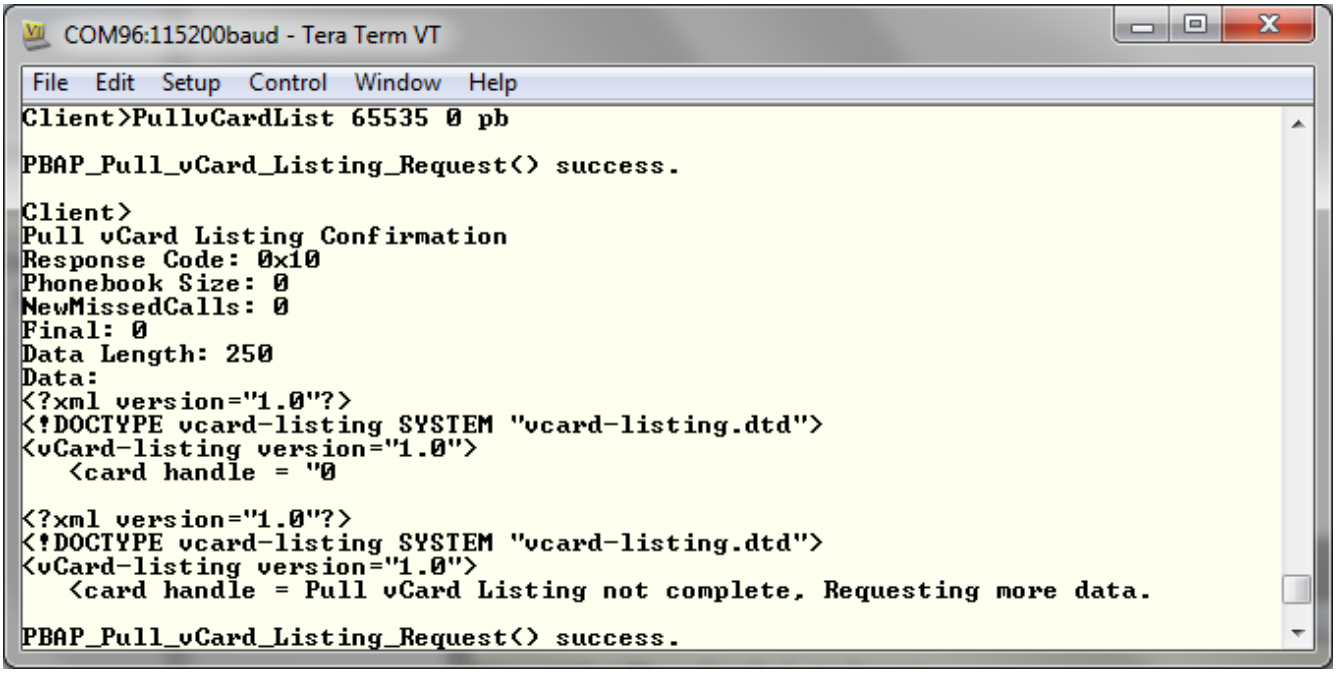

Figure 10-7. PBAP Demo Client Connection 3

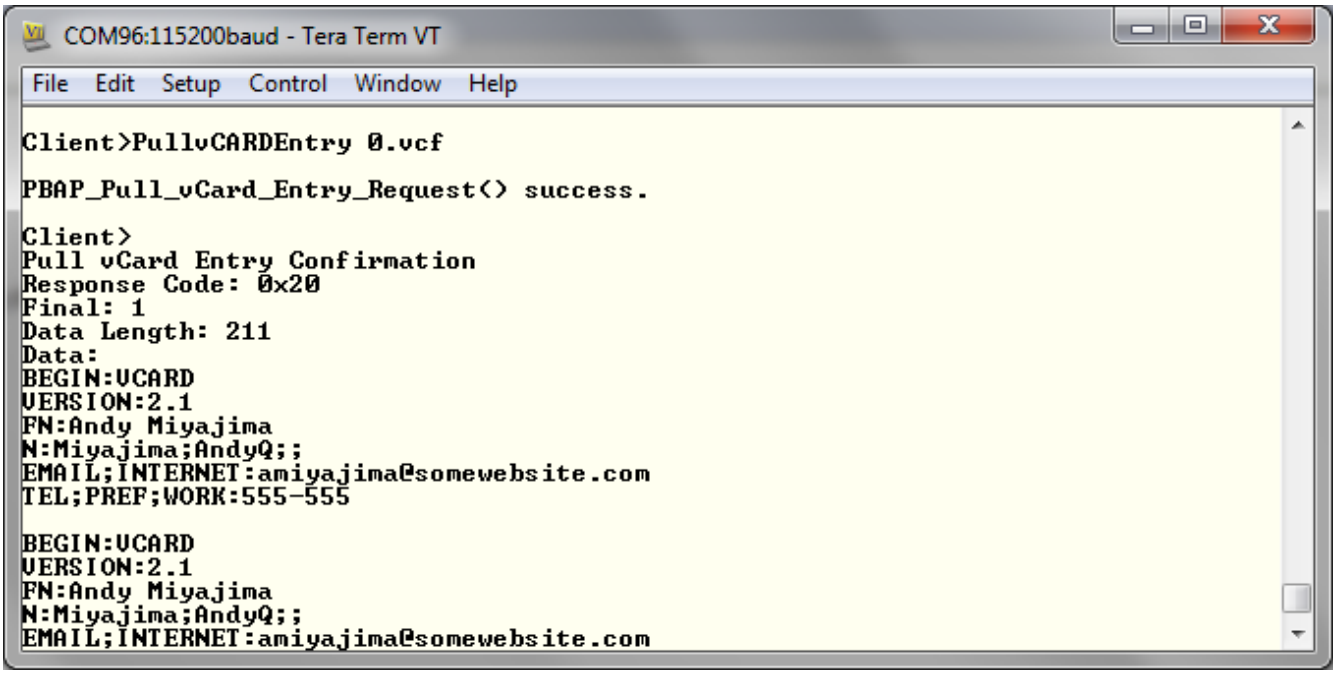

Figure 10-8. PBAP Demo Client Connection 4

10.3 Application Commands

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands

Phone Book Access Profile Commands

Open (OpenServer)

Description

The following function is responsible for creating a local PBAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

The command only requires one parameter. This parameter is the Port Number.

Command Call Examples

"Open 1" Attempts to Open a PBA Port Server at Port Number #1".

Possible Return Values

(0)PBAP_Register_Server_SDP_Record

(-4) FUNCTION_ERROR

(-9) UNABLE_TO_REGISTER_SERVER

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

(-1000) BTPBAP_ERROR_INVALID_PARAMETER

(-1001) BTPBAP_ERROR_NOT_INITIALIZED

(-1002) BTPBAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTPBAP_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTPBAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTPBAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTPBAP_ERROR_ACTION_NOT_ALLOWED

API Call

PBAP_Open_Server_Port(BluetoothStackID, TempParam->Params[0].intParam, (PBAP_SUPPORTED_REPOSITORIES_LOCAL_PHONEBOOK), PBAP_Event_Callback_Server, 0)

API Prototype

int BTPSAPI PBAP_Open_Server_Port(unsigned int BluetoothStackID, unsigned int ServerPort, Byte_t SupportedRepositories, PBAP_Event_Callback_t EventCallback, unsigned long CallbackParameter)

Description of API

The following function is responsible for opening a local PBAP Server. The first parameter is the Bluetooth Stack ID on which to open the Server. The second parameter is the Port on which to open this Server, and *MUST* be between PBAP_PORT_NUMBER_MINIMUM and PBAP_PORT_NUMBER_MAXIMUM. The third parameter is a bitmask which determines which repositories are supported by this Server instance. The fourth parameter is the Callback function to call when an event occurs on this Server Port. The final parameter is a user-defined callback parameter that will be passed to the callback function with each event. This function returns a positive, non zero value if successful or a negative return error code if an error occurs. A successful return code will be a PBAP Profile ID that can be used to reference the Opened PBAP Profile Server Port in ALL other PBAP Server functions in this module. Once an PBAP Profile Server is opened, it can only be Un-Registered via a call to the PBAP_Close_Server() function (passing the return value from this function).

Close

Description

The following function is responsible for deleting a local PBAP Server that was created via a successful call to the OpenServer() function. This function returns zero if successful and a negative value if an error occurred.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Server closing.

Possible Return Values

(0) Successfully Closed Server

(-4) FUNCTION_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

(-1000) BTPBAP_ERROR_INVALID_PARAMETER

(-1001) BTPBAP_ERROR_NOT_INITIALIZED

(-1002) BTPBAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTPBAP_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTPBAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTPBAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTPBAP_ERROR_ACTION_NOT_ALLOWED

API Call

PBAP_Close_Server_Port(BluetoothStackID, PBAPID);

API Prototype

int BTPSAPI PBAP_Close_Server_Port(unsigned int BluetoothStackID, unsigned int PBAPID)

Description of API

The following function is responsible for closing a PBAP Profile Server (which was opened by a successful call to the PBAP_Open_Server_Port() function). The first parameter is the Bluetooth Stack ID of the previously opened Server port. The second parameter is the PBAP ID returned from the previous call to PBAP_Open_Server_Port(). This function returns zero if successful, or a negative return error code if an error occurred (see BTERRORS.H). Note that this function does NOT delete any SDP Service Record Handles.

Client Commands
Open(OpenRemoteServer)

Description

The following function is responsible for initiating a connection with a Remote PBAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

The command takes two parameters to work. The first is the Inquiry Index which can be found using the DisplayInquiryList command after an Inquiry has been completed. The second is the RFCOMM Server Port which will be used to open a Remote PBAP Port.

Command Call Examples

"Open 12 4" Attempts to Open a Remote PhoneBook Port Server with the Remote Bluetooth Device whose address is found at the twelfth Inquiry Index using RFCOMM Server Port #4."

"Open 1 1" Attempts to Open a Remote PhoneBook Port Server with the Remote Bluetooth Device whose address is found at the first Inquiry Index using RFCOMM Server Port #1."

"Open 19 3" Attempts to Open a Remote PhoneBook Port Server with the Remote Bluetooth Device whose address is found at the nineteenth Inquiry Index using RFCOMM Server Port #3. "

Possible Return Values

(0) PBAP_Open_Remote_Server_Port: Function Successful

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-72) BTPS_ERROR_RFCOMM_UNABLE_TO_CONNECT_TO_REMOTE_DEVICE

(-73) BTPS_ERROR_RFCOMM_UNABLE_TO_COMMUNICATE_WITH_REMOTE_DEVICE

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

(-1000) BTPBAP_ERROR_INVALID_PARAMETER

(-1001) BTPBAP_ERROR_NOT_INITIALIZED

(-1002) BTPBAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTPBAP_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTPBAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTPBAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTPBAP_ERROR_ACTION_NOT_ALLOWED

API Call

PBAP_Open_Remote_Server_Port(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam-1)], TempParam->Params[1].intParam, PBAP_Event_Callback_Client, 0);

API Prototype

int BTPSAPI PBAP_Open_Remote_Server_Port(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, unsigned int ServerPort, PBAP_Event_Callback_t EventCallback, unsigned long CallbackParameter)

Description of API

The following function is responsible for opening a connection to a remote PBAP Server. The first parameter is the Bluetooth Stack ID of the local Bluetooth stack. The second parameter is the remote Bluetooth Device Address of the Bluetooth PBAP Profile Server with which to connect. The third parameter specifies the remote Server port with which to connect. The final two parameters specify the PBAP Profile Event Callback Function and the Callback Parameter to associate with this PBAP Profile Client. The ServerPort parameter *MUST* be between PBAP_PORT_NUMBER_MINIMUM and PBAP_PORT_NUMBER_MAXIMUM. This function returns a positive, non zero, value if successful or a negative return error code if an error occurs. A successful return code will be a PBAP ID that can be used to reference the remote opened PBAP Profile Server in ALL other PBAP Profile Client functions in this module. Once a remote Server is opened, it can only be closed via a call to the PBAP_Close_Connection() function (passing the return value from this function).

Close(CloseConnection)

Description

The following function is responsible for terminating a connection with a remote PBAP Client or Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Server closing.

Possible Return Values

(0) PBAP_Open_Remote_Server_Port: Function Successful

(-4) FUNCTION_ERROR

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-73) BTPS_ERROR_RFCOMM_UNABLE_TO_COMMUNICATE_WITH_REMOTE_DEVICE

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

(-1000) BTPBAP_ERROR_INVALID_PARAMETER

(-1001) BTPBAP_ERROR_NOT_INITIALIZED

(-1002) BTPBAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTPBAP_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTPBAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTPBAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTPBAP_ERROR_ACTION_NOT_ALLOWED

API Call

PBAP_Close_Connection(BluetoothStackID, PBAPID);

API Prototype

BTPSAPI PBAP_Close_Connection(unsigned int BluetoothStackID, unsigned int PBAPID)

Description of API

The following function is responsible for closing a currently ongoing PBAP Profile connection. The first parameter is the Bluetooth Stack ID of the Bluetooth Protocol Stack Instance that is associated with the PBAP Profile connection being closed. The second parameter to this function is the PBAP ID of the PBAP Profile connection to be closed. This function returns zero if successful, or a negative return value if there was an error.

Note

If this function is called with a Server PBAP ID (a value returned from a call to PBAP_Open_Server_Port()) any clients currently connected to this Server will be terminated, but the Server will remained open and registered. If this function is called using a Client PBAP ID (a value returned from a call to PBAP_Open_Remote_Server_Port()), the Client connection will be terminated/closed entirely.

OBEXDisconnect

Description

The following function is responsible for terminating a connection with a remote PBAP Server by issuing an OBEX Disconnect. This function returns zero if successful and a negative value if an error occurred.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Server closing.

Possible Return Values

(0) PBAP_Disconnect_Request: Function Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-73) BTPS_ERROR_RFCOMM_UNABLE_TO_COMMUNICATE_WITH_REMOTE_DEVICE

(-1000) BTPBAP_ERROR_INVALID_PARAMETER

(-1001) BTPBAP_ERROR_NOT_INITIALIZED

(-1002) BTPBAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTPBAP_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTPBAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTPBAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTPBAP_ERROR_ACTION_NOT_ALLOWED

API Call

PBAP_Disconnect_Request(BluetoothStackID, PBAPID);

API Prototype

int BTPSAPI PBAP_Disconnect_Request(unsigned int BluetoothStackID, unsigned int PBAPID)

Description of API

The following function is responsible for sending an Disconnect Request to the remote device/entity. The first parameter to this function is the Bluetooth Stack ID of the Bluetooth Protocol Stack Instance that is associated with the PBAP Profile connection being disconnected. The second parameter to this function is the PBAP ID of the PBAP Profile connection to be disconnected (that was returned by a call to PBAP_Open_Remote_Server_Port()). This function returns zero if successful, or a negative return value if there was an error.

Note

Use of this function is optional and is provided for full compliance with some OBEX applications. Calling the PBAP_Close_Connection() function will achieve the same results without sending the OBEX disconnect packet beforehand. If is also possible to call this function and then immediately call PBAP_Close_Connection() without waiting for a confirmation because a Disconnect Request cannot be failed. Calling this function by itself and waiting for a response will cause the underlying connection to automatically be closed once the response is received. This will generate a Close Port Indication.

PullPhonebook

Description

The following function is responsible for issuing a PBAP Pull Phonebook Request with a remote PBAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

The command requires 3 parameters, the Max List Count, the List Start Offset and the Phonebook Path.

Command Call Examples

"PullPhonebook 65535 0 telecom/pb.vcf has 65535 as the Max List Count, 0 as the List Start Offset and telecom/pb.vcf as the Phonebook Path.

Possible Return Values

(0) PBAP_Pull_Phonebook_Request: Function Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTPBAP_ERROR_INVALID_PARAMETER

(-1001) BTPBAP_ERROR_NOT_INITIALIZED

(-1002) BTPBAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTPBAP_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTPBAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTPBAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTPBAP_ERROR_ACTION_NOT_ALLOWED

API Call

PBAP_Set_Phonebook_Request(BluetoothStackID, PBAPID, (PBAP_Set_Path_Option_t)TempParam->Params[0].intParam, TempParam->Params[1].strParam);

API Prototype

*int BTPSAPI PBAP_Pull_Phonebook_Request(unsigned int BluetoothStackID, unsigned int PBAPID, char *ObjectName, DWord_t FilterLow, DWord_t FilterHigh, PBAP_Format_t Format, Word_t MaxListCount, Word_t ListStartOffset)*

Description of API

The following function generates a PBAP Pull Phonebook Request to the specified remote PBAP Server. The BluetoothStackID parameter the ID of the Bluetooth Stack that is associated with this PBAP Client. The PBAPID parameter specifies the PBAP ID for the local PBAP Client (returned from a successful call to the PBAP_Connect_Remote_Server_Port() function). The ObjectName parameter contains the Name/Path of the Phonebook being requested by this Pull Phonebook operation. The FilterLow parameter contains the lower 32 bits of the 64-bit filter attribute. The FilterHigh parameter contains the higher 32 bits of the 64-bit filter attribute. The Format parameter is an enumeration which specifies the vCard format requested in this Pull Phonebook request. If pfDefault is specified then the format will not be included in the request (note that the Server will default to pfvCard21 in this case). The MaxListCount parameter is an unsigned integer that specifies the maximum number of entries the Client can handle. A value of 65535 means that the number of entries is not restricted. A MaxListCount of ZERO (0) indicates that this is a request for the number of used indexes in the Phonebook specified by the ObjectName parameter. The ListStartOffset parameter specifies the index requested by the Client in this PullPhonebookRequest. This function returns zero if successful or a negative return error code if there was an error.

Note

A successful return code does not mean that the remote PBAP Profile Server successfully processed the command. The caller needs to check the confirmation result to determine if the remote PBAP Profile Server successfully executed the Request. There can only be one outstanding PBAP Profile Request active at any one time. Because of this, another PBAP Profile Request cannot be issued until either the current request is Aborted (by calling the PBAP_Abort_Request() function) or the current Request is completed (this is signified by receiving a Confirmation Event in the PBAP Profile Event Callback that was registered when the PBAP Profile Port was opened).

SetPhonebook

Description

The following function is responsible for issuing a PBAP Set Phonebook Request with a remote PBAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

This functions requires two parameters; the path change requested and the path name. E.g If down is selected as the path option then we require an additional paramters which is the path.

Command Call Examples

"SetPhonebook 1 telecom goes down a level to the telecom folder.

Possible Return Values

(0) PBAP_Set_Phonebook_Request: Function Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTPBAP_ERROR_INVALID_PARAMETER

(-1001) BTPBAP_ERROR_NOT_INITIALIZED

(-1002) BTPBAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTPBAP_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTPBAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTPBAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTPBAP_ERROR_ACTION_NOT_ALLOWED

API Call

PBAP_Set_Phonebook_Request(BluetoothStackID, PBAPID, (PBAP_Set_Path_Option_t)TempParam->Params[0].intParam, TempParam->Params[1].strParam);

API Prototype

*int BTPSAPI PBAP_Set_Phonebook_Request(unsigned int BluetoothStackID, unsigned int PBAPID, PBAP_Set_Path_Option_t PathOption, char *ObjectName)*

Description of API

The following function generates a PBAP Set Phonebook Request to the specified remote PBAP Server. The BluetoothStackID parameter the ID of the Bluetooth Stack that is associated with this PBAP Client. The PBAPID parameter specifies the PBAP ID for the local PBAP Client (returned from a successful call to the PBAP_Connect_Remote_Server_Port() function). The PathOption parameter contains an enumerated value that indicates the type of path change to request. The ObjectName parameter contains the folder name to include with this Set Phonebook request. This value can be NULL if no name is required for the selected PathOption. See the PBAP specification for more information. This function returns zero if successful or a negative return error code if there was an error.

Note

A successful return code does not mean that the remote PBAP Profile Server successfully processed the command. The caller needs to check the confirmation result to determine if the remote PBAP Profile Server successfully executed the Request. There can only be one outstanding PBAP Profile Request active at any one time. Because of this, another PBAP Profile Request cannot be issued until either the current request is Aborted (by calling the PBAP_Abort_Request() function) or the current Request is completed (this is signified by receiving a Confirmation Event in the PBAP Profile Event Callback that was registered when the PBAP Profile Port was opened).

PullvCardList

Description

The following function is responsible for issuing a PBAP Pull vCARD Listing Request with a remote PBAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

The command requires 3 parameters, the Max List Count, the List Start Offset and the Phonebook Path.

Command Call Examples

"PullPhonebook 65535 0 pb has 65535 as the Max List Count, 0 as the List Start Offset and pb as the Phonebook Path. "

Possible Return Values

(0) PBAP_Pull_vCard_Listing_Request: Function Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTPBAP_ERROR_INVALID_PARAMETER

(-1001) BTPBAP_ERROR_NOT_INITIALIZED

(-1002) BTPBAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTPBAP_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTPBAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTPBAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTPBAP_ERROR_ACTION_NOT_ALLOWED

API Call

PBAP_Pull_vCard_Listing_Request(BluetoothStackID, PBAPID, Phonebook, loDefault, saDefault, NULL, (Word_t)TempParam->Params[0].intParam, (Word_t)TempParam->Params[1].intParam);

API Prototype

*int BTPSAPI PBAP_Pull_vCard_Listing_Request(unsigned int BluetoothStackID, unsigned int PBAPID, char *ObjectName, PBAP_List_Order_t ListOrder, PBAP_Search_Attribute_t SearchAttribute, char *SearchValue, Word_t MaxListCount, Word_t ListStartOffset)*

Description of API

The following function generates a PBAP Pull vCard Listing Request to the specified remote PBAP Server. The BluetoothStackID parameter the ID of the Bluetooth Stack that is associated with this PBAP Client. The PBAPID parameter specifies the PBAP ID for the local PBAP Client (returned from a successful call to the PBAP_Connect_Remote_Server_Port() function). The ObjectName parameter contains the folder of the Phonebook being requested by this Pull vCard Listing operation. This value can be NULL if a PhonebookSize request is being performed. The ListOrder parameter is an enumerated type that determines the optionally requested order of the listing. Using the 'loDefault' value for this parameter will prevent this field from being added to the request (note that the Server will default to loIndexed in this case). The SearchAttribute is an enumerated type that determines the optionally requested attribute used to filter this request. Using the 'saDefault' value for this parameter will prevent this field from being added to the request (note that the Server will default to saIndexed in this case). The SearchValue parameter contains an optional ASCII, Null-terminated character string that contains the string requested for search/filter. If this parameter is NULL, this field will be excluded from the request. The MaxListCount parameter is an unsigned integer that specifies the maximum number of list entries the Client can handle. A value of 65535 means that the number of entries is not restricted. A MaxListCount of ZERO (0) indicates that this is a request for the number of used indexes in the Phonebook specified by the ObjectName parameter. The ListStartOffset parameter specifies the index requested by the Client in this Pull vCard Listing. This function returns zero if successful or a negative return error code if there was an error.

Note

A successful return code does not mean that the remote PBAP Profile Server successfully processed the command. The caller needs to check the confirmation result to determine if the remote PBAP Profile Server successfully executed the Request. There can only be one outstanding PBAP Profile Request active at any one time. Because of this, another PBAP Profile Request cannot be issued until either the current request is Aborted (by calling the PBAP_Abort_Request() function) or the current Request is completed (this is signified by receiving a Confirmation Event in the PBAP Profile Event Callback that was registered when the PBAP Profile Port was opened).

PullvCARDEntry

Description

The following function is responsible for issuing a PBAP Pull vCARD Entry Request with a remote PBAP Server. This function returns zero if successful and a negative value if an error occurred.

Parameters

This function requires only one parameters, the vCard Name.

Command Call Examples

"PullvCARDEntry 0.vcf pulls the vcf file named 0."

Possible Return Values

(0) PBAP_Pull_vCard_Entry_Request: Function Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BTPBAP_ERROR_INVALID_PARAMETER

(-1001) BTPBAP_ERROR_NOT_INITIALIZED

(-1002) BTPBAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTPBAP_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTPBAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTPBAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTPBAP_ERROR_ACTION_NOT_ALLOWED

API Call

PBAP_Pull_vCard_Entry_Request(BluetoothStackID, PBAPID, TempParam->Params[0].strParam, 0, 0, pfvCard21);

API Prototype

*int BTPSAPI PBAP_Pull_vCard_Entry_Request(unsigned int BluetoothStackID, unsigned int PBAPID, char *ObjectName, DWord_t FilterLow, DWord_t FilterHigh, PBAP_Format_t Format)*

Description of API

The following function generates a PBAP Pull vCard Entry Request to the specified remote PBAP Server. The BluetoothStackID parameter the ID of the Bluetooth Stack that is associated with this PBAP Client. The PBAPID parameter specifies the PBAP ID for the local PBAP Client (returned from a successful call to the PBAP_Connect_Remote_Server_Port() function). The ObjectName parameter contains the name of the Phonebook entry being requested by this Pull vCard Entry operation. The FilterLow parameter contains the lower 32 bits of the 64-bit filter attribute. The FilterHigh parameter contains the higher 32 bits of the 64-bit filter attribute. The Format parameter is an enumeration which specifies the vCard format requested in this Pull vCard Entry request. If pfDefault is specified then the format will not be included in the request (note that in this case the Server will default to pfvCard21 in this case). This function returns zero if successful or a negative return error code if there was an error.

Note

A successful return code does not mean that the remote PBAP Profile Server successfully processed the command. The caller needs to check the confirmation result to determine if the remote PBAP Profile Server successfully executed the Request. There can only be one outstanding PBAP Profile Request active at any one time. Because of this, another PBAP Profile Request cannot be issued until either the current request is Aborted (by calling the PBAP_Abort_Request()function) or the current Request is completed (this is signified by receiving a Confirmation Event in the PBAP Profile Event Callback that was registered when the PBAP Profile Port was opened).

Abort

Description

The following function is responsible for issuing an Abort with a remote PBAP Server by issuing an OBEX Abort Command. This function returns zero if successful and a negative value if an error occurred.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Server closing.

Possible Return Values

(0) PBAP_Abort_Request: Function Successful

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-1000) BTPBAP_ERROR_INVALID_PARAMETER

(-1001) BTPBAP_ERROR_NOT_INITIALIZED

(-1002) BTPBAP_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003) BTPBAP_ERROR_LIBRARY_INITIALIZATION_ERROR

(-1004) BTPBAP_ERROR_INSUFFICIENT_RESOURCES

(-1005) BTPBAP_ERROR_REQUEST_ALREADY_OUTSTANDING

(-1006) BTPBAP_ERROR_ACTION_NOT_ALLOWED

API Call

PBAP_Abort_Request(BluetoothStackID, PBAPID);

API Prototype

int BTPSAPI PBAP_Abort_Request(unsigned int BluetoothStackID, unsigned int PBAPID)

Description of API

The following function is responsible for Aborting ANY currently outstanding PBAP Profile Client Request. The first parameter is the Bluetooth Stack ID of the Bluetooth Stack for which the PBAP Profile Client is valid. The second parameter to this function specifies the PBAP ID (returned from a successful call to the PBAP_Open_Remote_Server_Port() function). This function returns zero if successful, or a negative return error code if there was an error.

Note

There can only be one outstanding PBAP Profile Request active at any one time. Because of this, another PBAP Profile Request cannot be issued until either the current request is Aborted (by calling the PBAP_Abort_Request() function) or the current Request is completed (this is signified by receiving a Confirmation Event in the PBAP Profile Event Callback that was registered when the PBAP Profile Port was opened). Because of transmission latencies, it may be possible that a PBAP Profile Client Request that is to be aborted may have completed before the Server was able to Abort the request. In either case, the caller will be notified via PBAP Profile Callback of the status of the previous Request.

11 SPP Demo Guide

11.1 Demo Overview

Note

The same instructions can be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms.

This demo allows users to evaluate TI's CC256x Bluetooth device by using the PAN1323EMK, MSP-EXP430F5438 board, Tiva DK-TM4C129X, MSP432 or STM32F4. The SPP sample application code is provided to enable a rich out-of-box experience to the user. The application allows the user to use a console to send Bluetooth commands, setup a Bluetooth Device to accept connections, connect to a remote Bluetooth device and communicate over Bluetooth.

For information of the LE version of this profile, refer to the document SPP-LE profile.

It is recommended that the user visits the kit setup Getting Started Guide for MSP430, Getting Started Guide for TIVA, Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as **MSP-EXP430F5438 USB - Serial Port(COM x)**, **Tiva Virtual COM Port (COM x)**, **XDS110 Class Application/User UART (COM x)** for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud (9600 for MSP430), 8, n, 1. Once connected, reset the device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

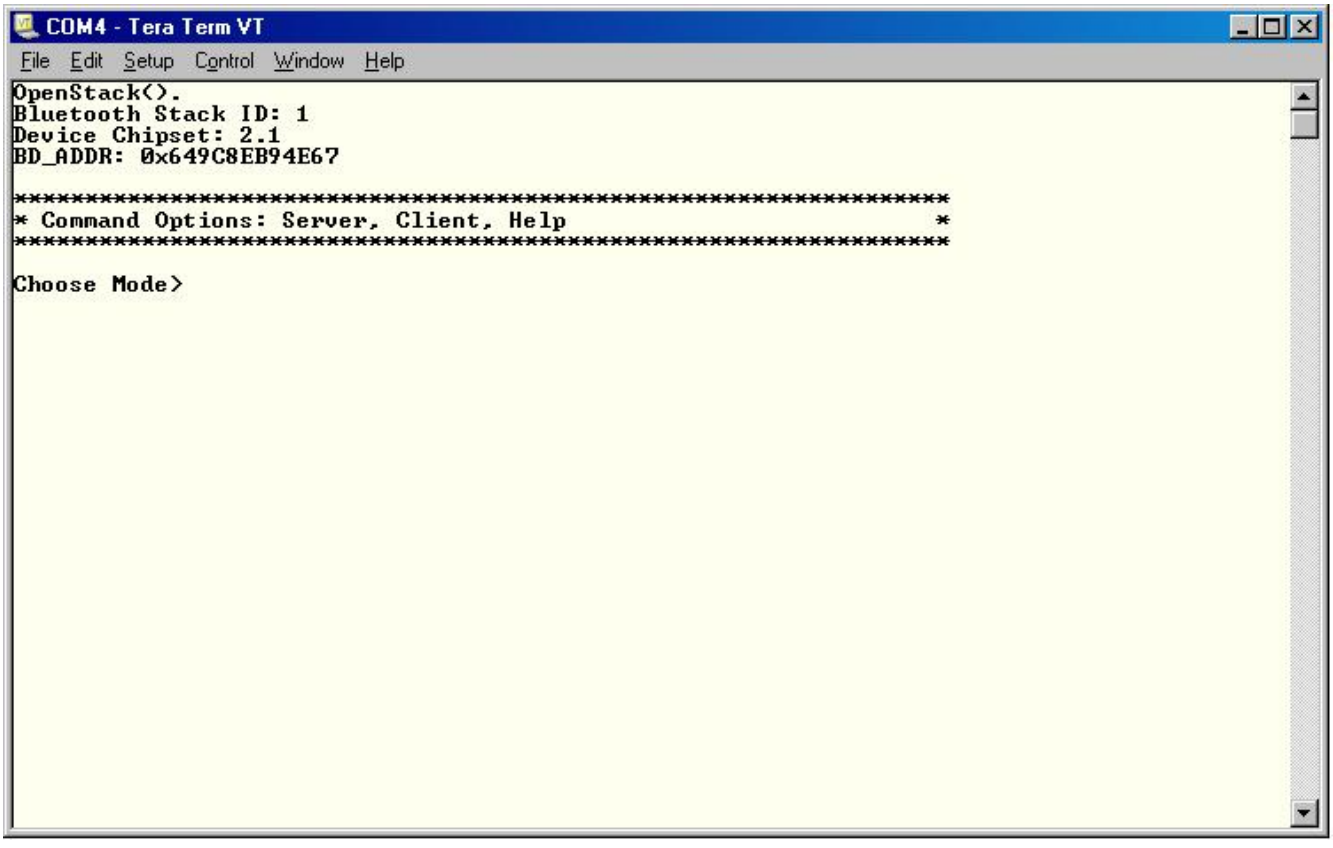

Figure 11-1. SPP Demo Initial Screen of SPP Demo Application

11.2 Demo Application

This section provides a description of how to use the demo application to connect two configured board and communicate over Bluetooth. Bluetooth SPP is a simple Client-Server connection process. We will setup one of the boards as a server and the other board as a client. We will then initiate a connection from the client to the server. Once connected, we can transmit data between the two devices over Bluetooth.

Server setup on the demo application

1. We will setup the first board as a server. Perform the steps mentioned earlier in "Running the Bluetooth Code" section to initialize the application. Once initialized, note the Bluetooth address of the server. We will later use this to initiate a connection from the client.
2. On the "Choose mode>" prompt, enter **Server**.
3. You will see a list of all possible commands at this time for a server. You can see this list at any time by entering **Help** at the Server> prompt.
4. Now we are ready to open a server. To open a server, at the "Server>" prompt, enter **Open 1**. You can replace 1 with any number between 1 and 30, as long as there is no server open on that port. Once you see "Server opened: 1", you have a SPP server open on port 1.

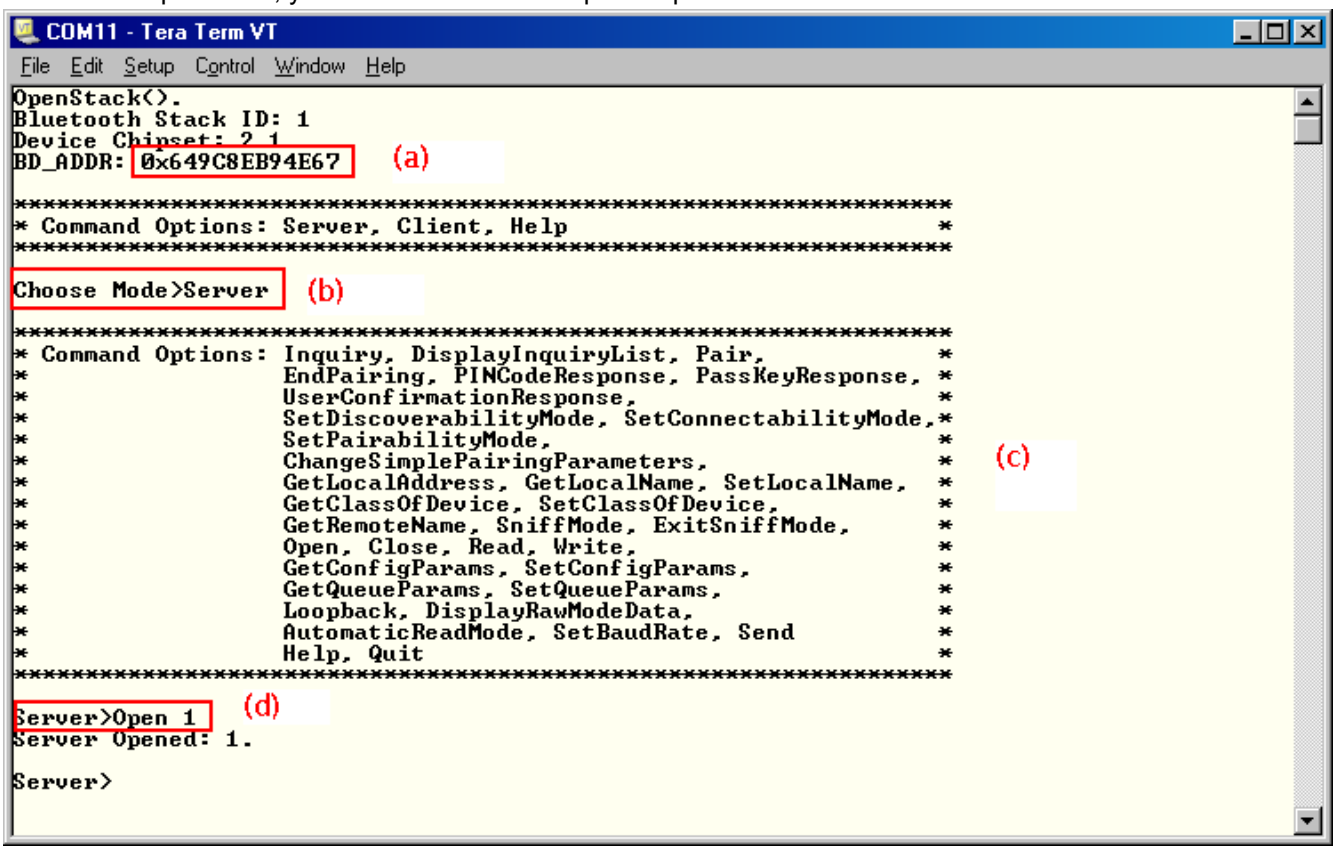

Figure 11-2. SPP Demo Server Setup

Client setup and device discovery on the demo application

1. We will setup the second board as a client. Perform the steps mentioned earlier in "Running the Bluetooth Code" section to initialize the application. On the "Choose mode>" prompt, enter **Client**.
2. You will see a list of all possible commands at this time for a Client. You can see this list at any time by entering **Help** at the Client> prompt.
3. At the "Client>" prompt, enter **Inquiry**. This will initiate the Inquiry process. Once it is complete, you will get a list of all discovered devices.
4. You can access this list any time by choosing **DisplayInquiryList** at the Client prompt.

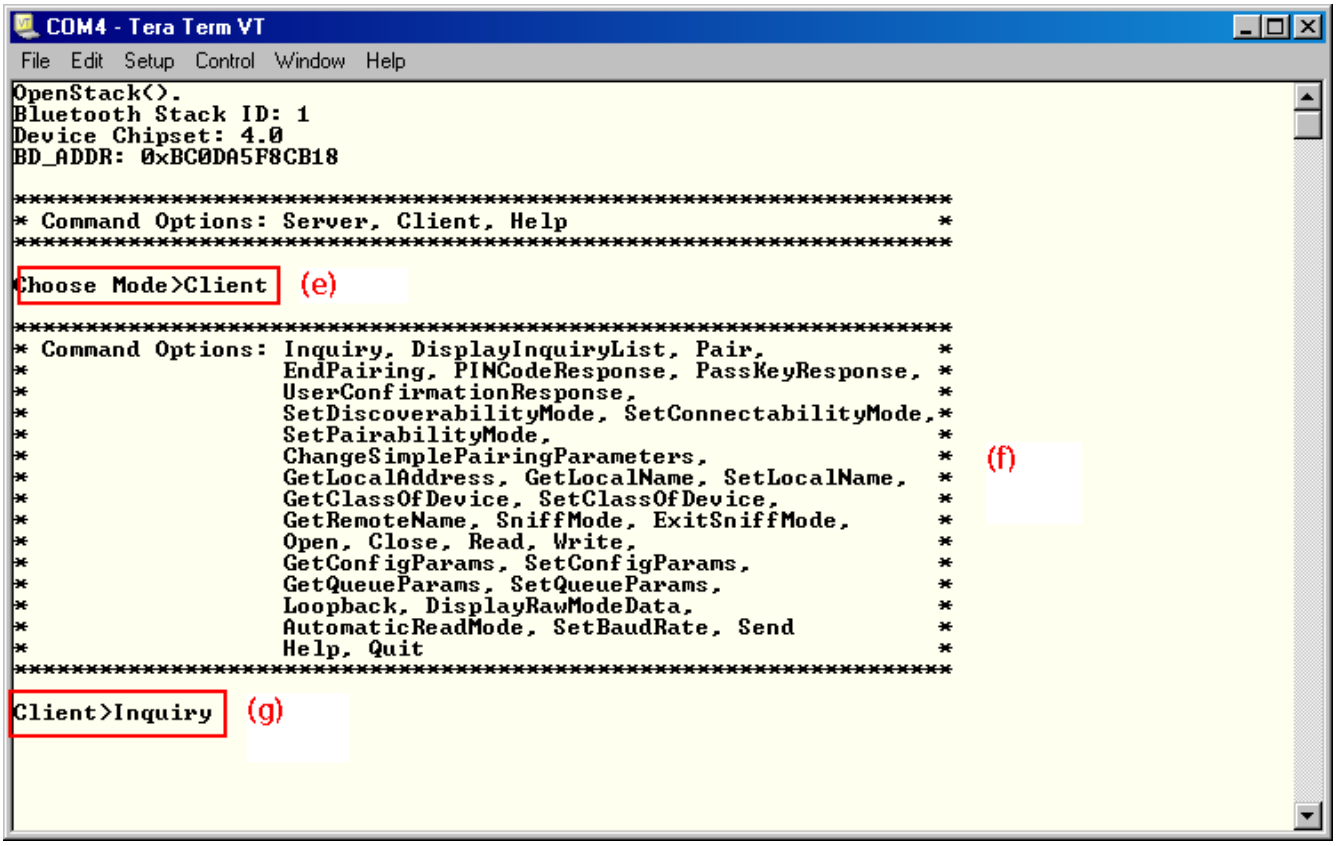

Figure 11-3. SPP Demo Client Setup

Initiating connection from the client

1. Note the index number of the first board that was configured as a server. [If the list is not on the screen, issue **DisplayInquiryList** command on the client to display the list of discovered devices again.]
2. Issue a **Open <index number> <server port number>** command at the command prompt.
3. Wait for SPP Open confirmation. I) When a client successfully connects to a server, the server will see the open indication.

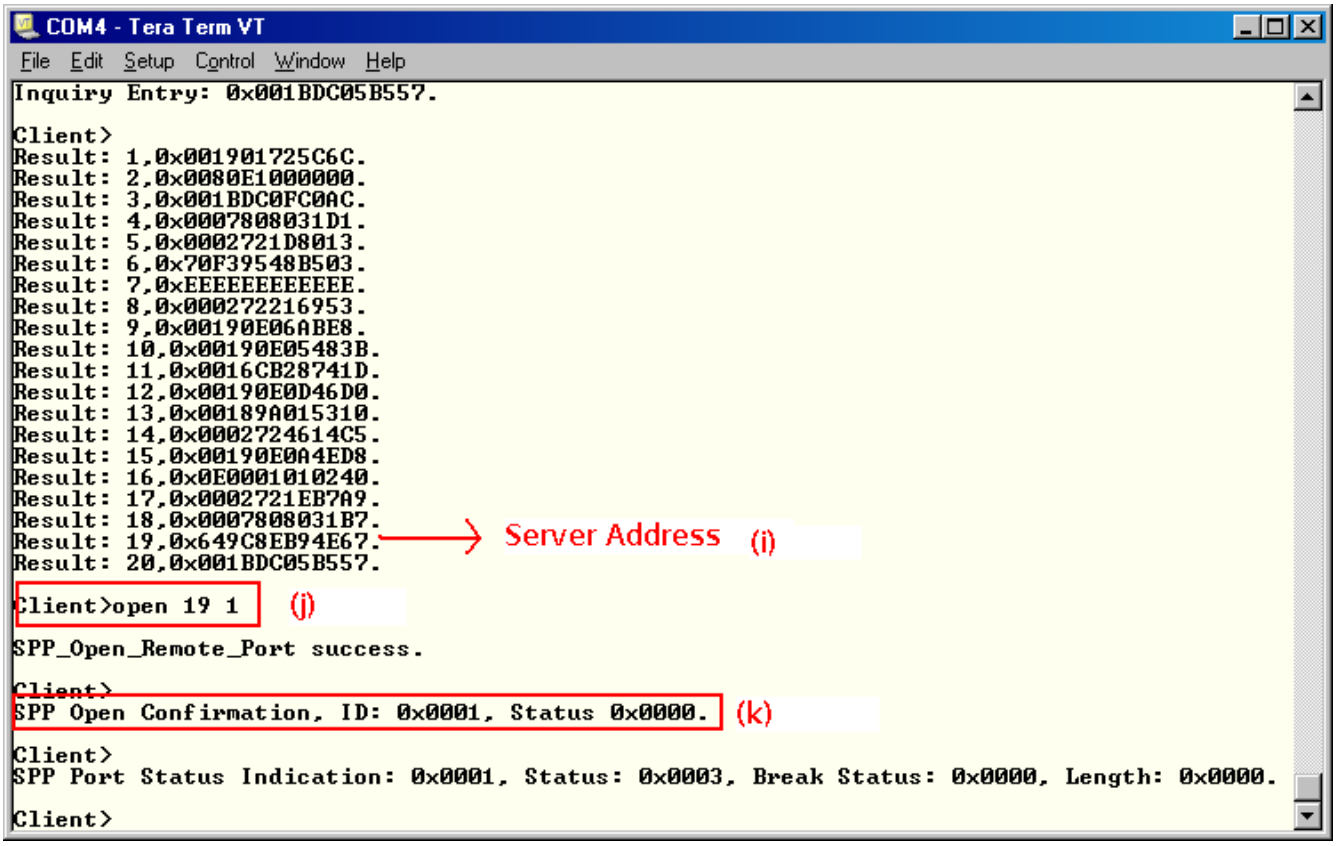

Figure 11-4. SPP Demo Client Connection

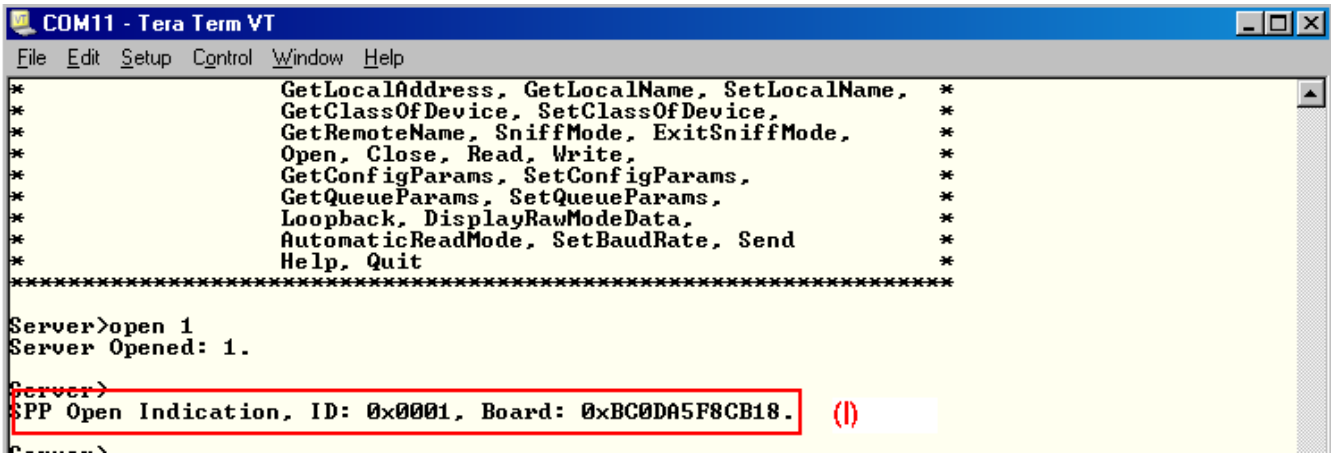

Figure 11-5. SPP Demo Server Connection

Data Transfer between Client and Server

1. Now we have a SPP connection established and both devices are ready to transmit data to each other.
2. On Client or Server you can send some data to the remote side by issuing a **Write** command. This command sends a hardcoded test string to the other side.
3. The remote side will receive a data indication
4. The user can read the data by issuing a **Read** command.
5. The connection can be closed on either side by issuing the close command. In the example the client closes the connection and the server receives a close indication.

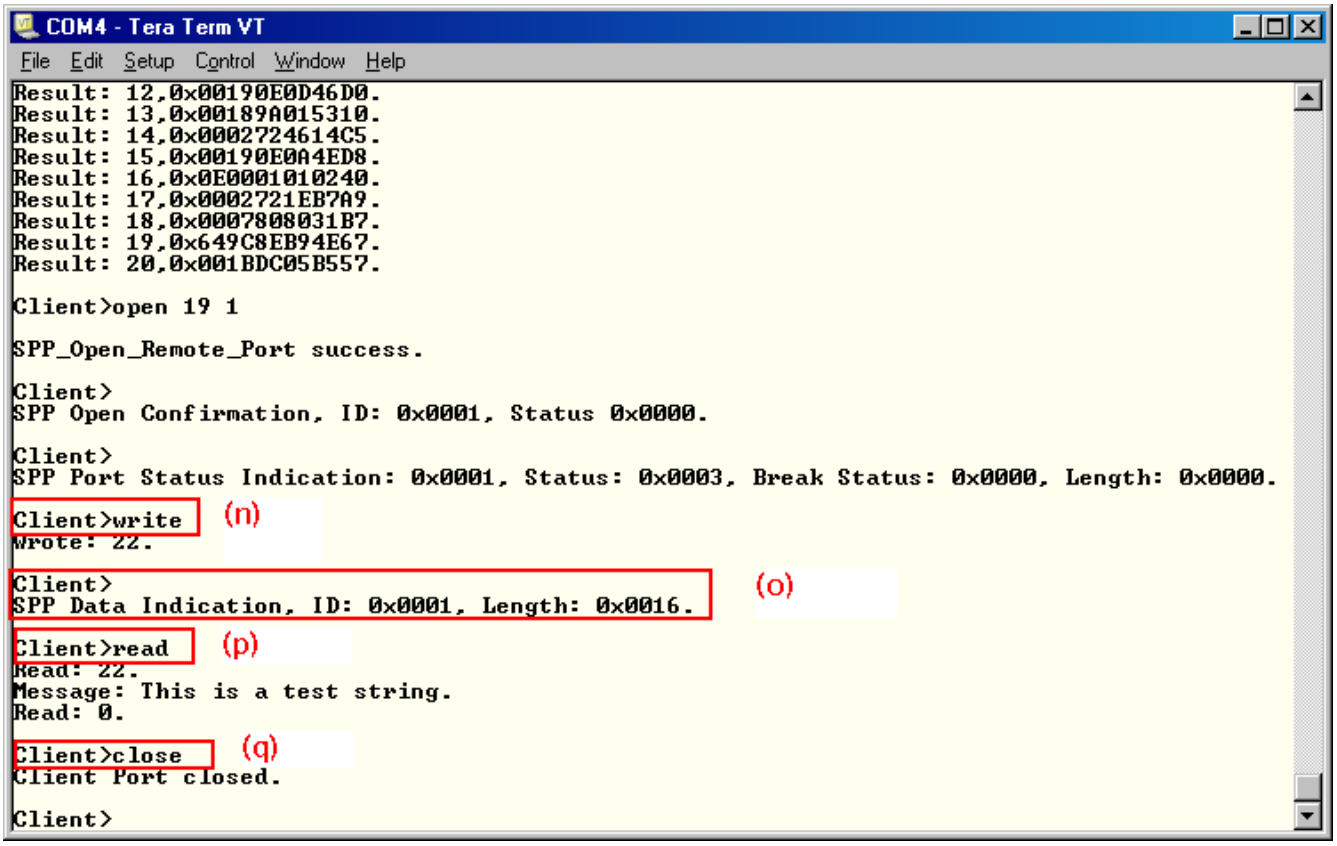

Figure 11-6. SPP Demo Client Connection and Data Transfer

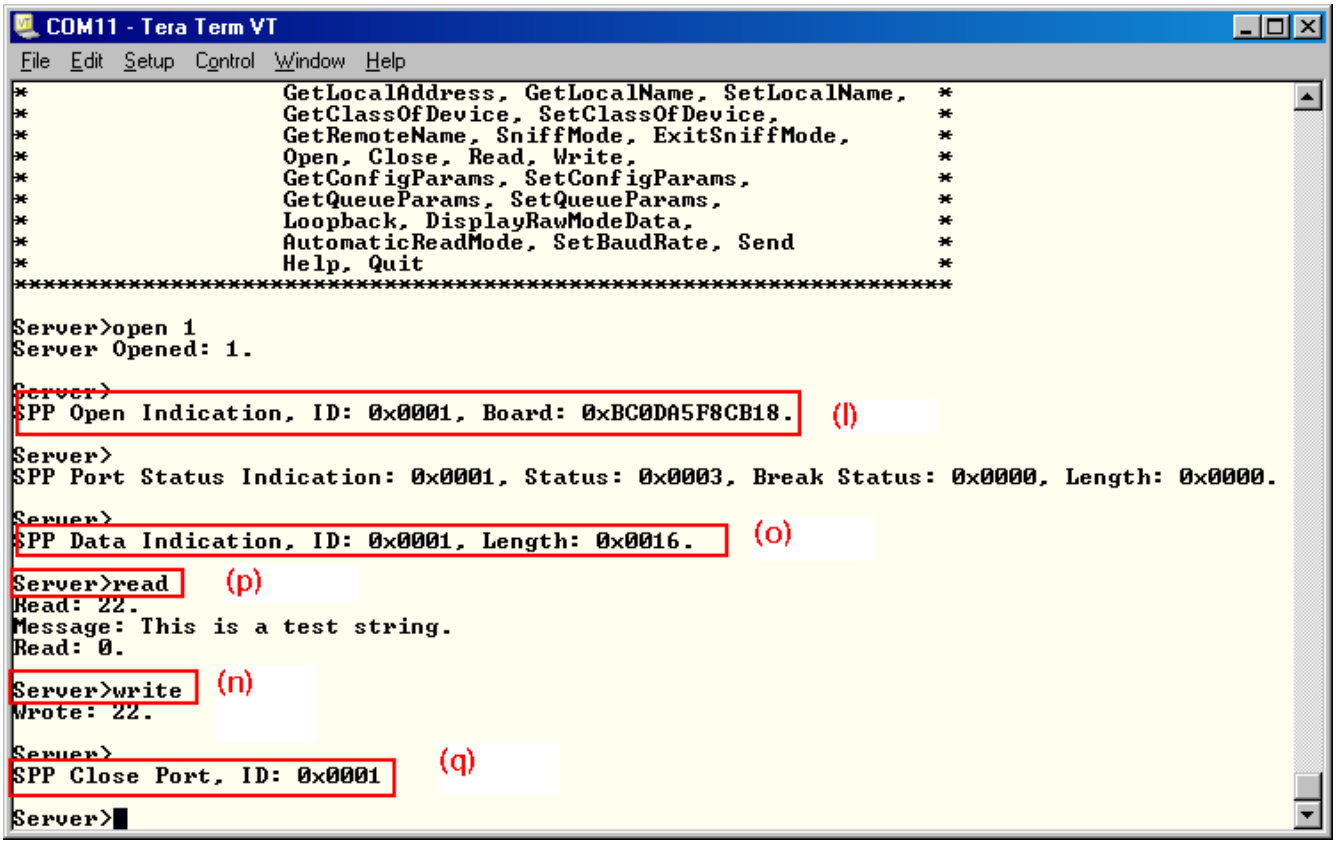

Figure 11-7. SPP Demo Server Connection and Data Transfer

Example connection using Blueterm

We will demonstrate a SPP connection using Blueterm. Blueterm is an app that can be used to connect an Android device to the SPPDemo. For more about the app refer to BlueTerm-Google Play.

Note

The app requires Android V2.1 and above.

1. Open a server port on the MSP430.

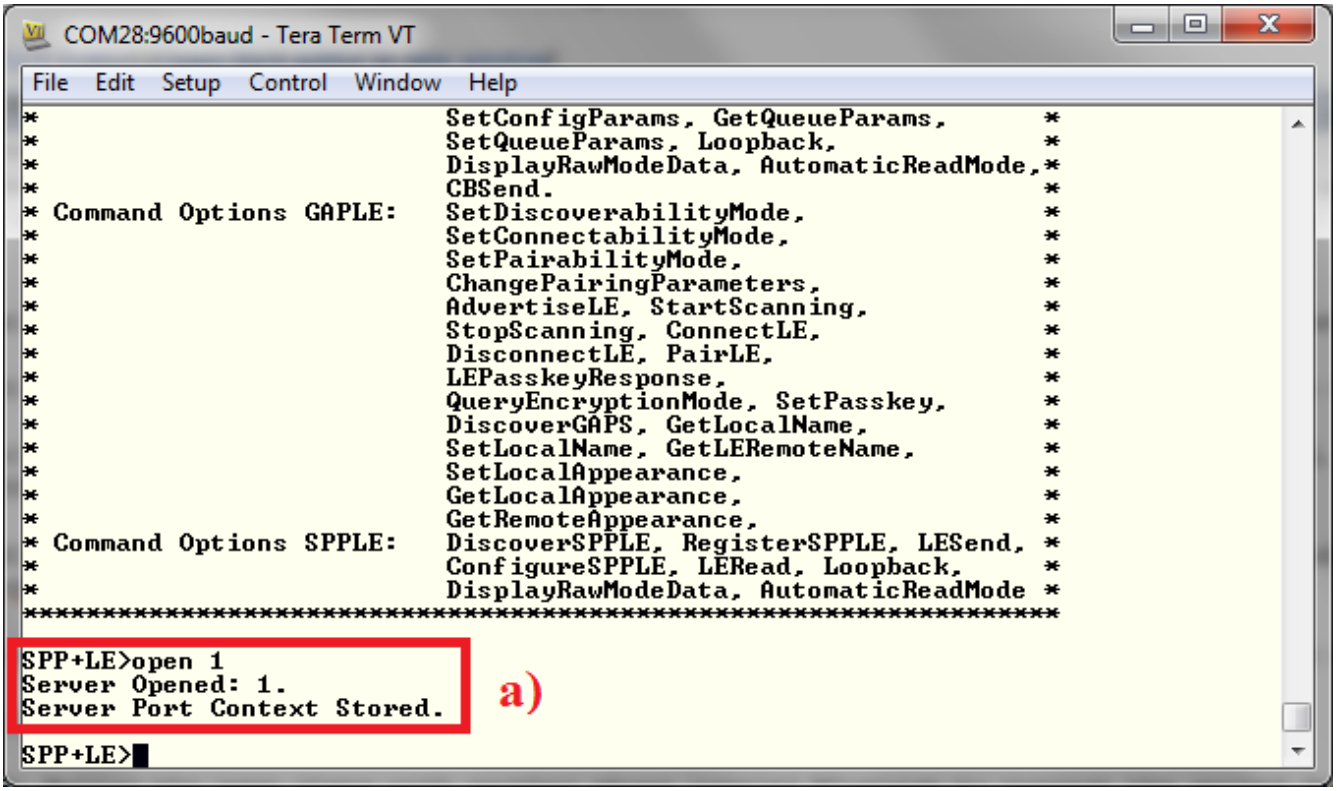

Figure 11-8. SPP Demo Multiple SPP

2. To connect to the device, open blueterm, press the menu button and hit the **connect** icon.

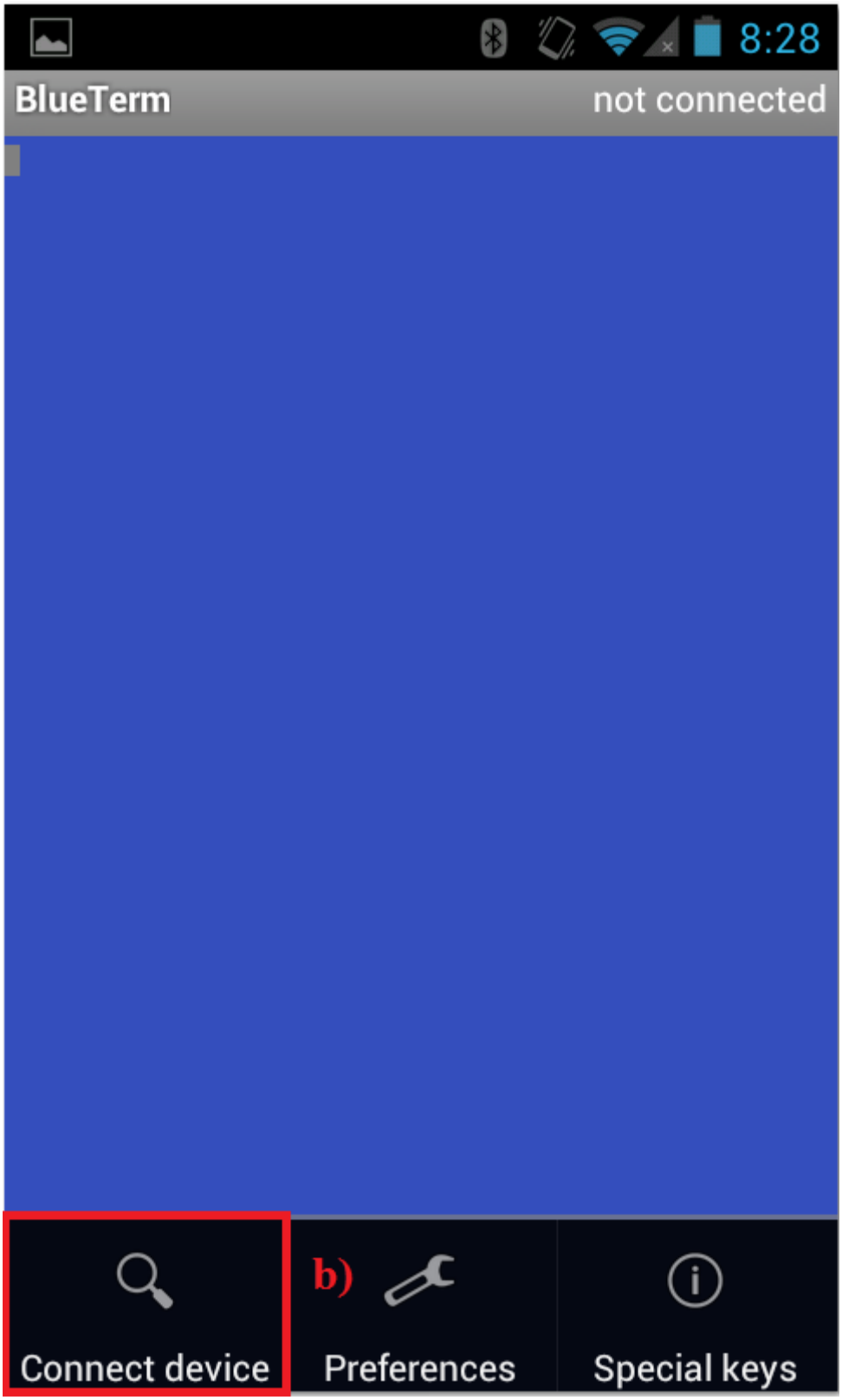

Figure 11-9. SPP Demo Connect on Blueterm

3. The app should show a list of paired devices. If the device is not already paired, select scan. It should search for available bluetooth devices.

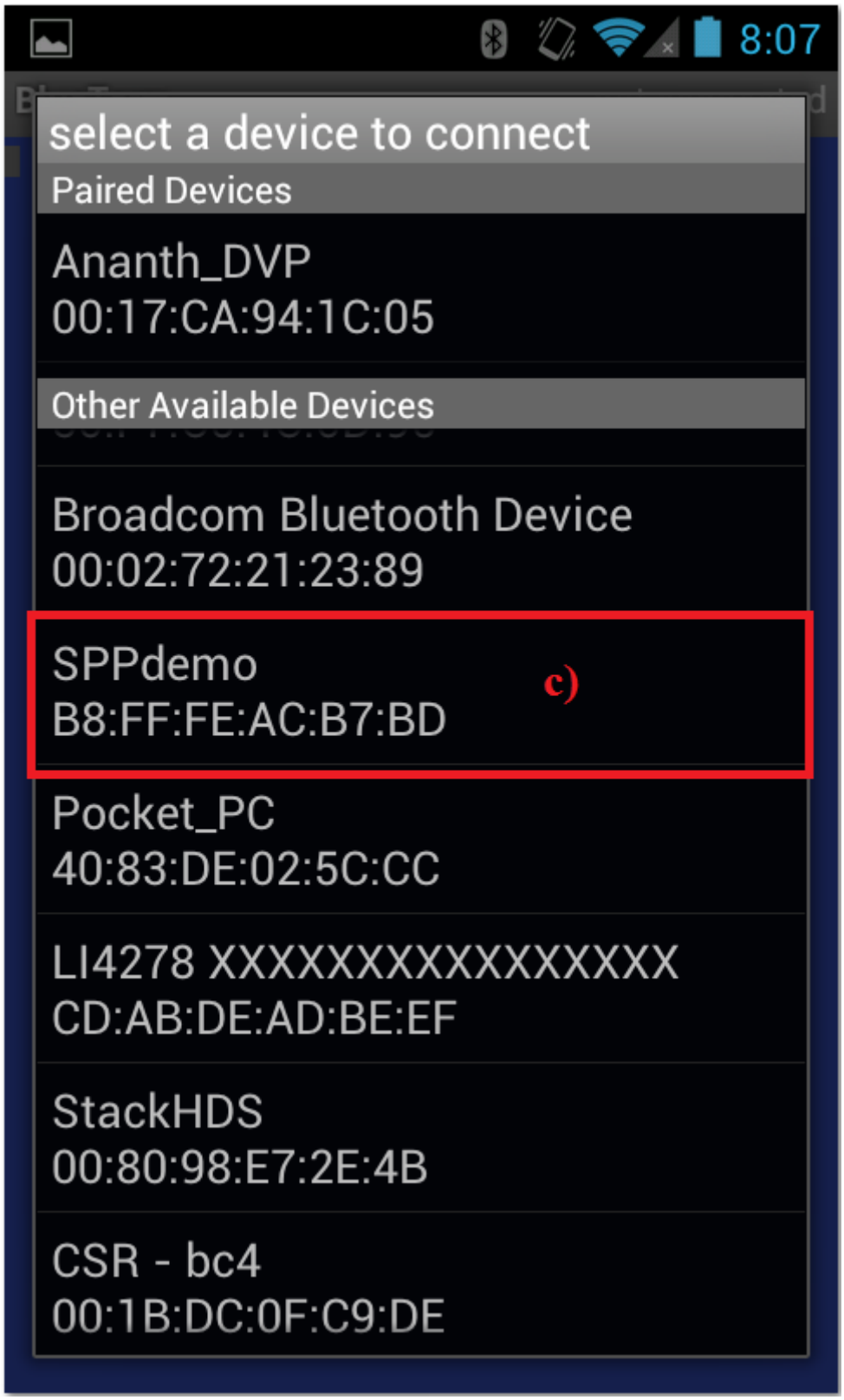

Figure 11-10. SPP Demo Available devices

4. Select and Pair it with the device running the SPPdemo. If it needs a pincode enter a pincode on the app/phone. The MSP430 will prompt for a **PINCodeResponse**. Type **PINCodeResponse <the pin code you used for the phone>** in the prompt. It will pair and you should see an open indication on the MSP430.

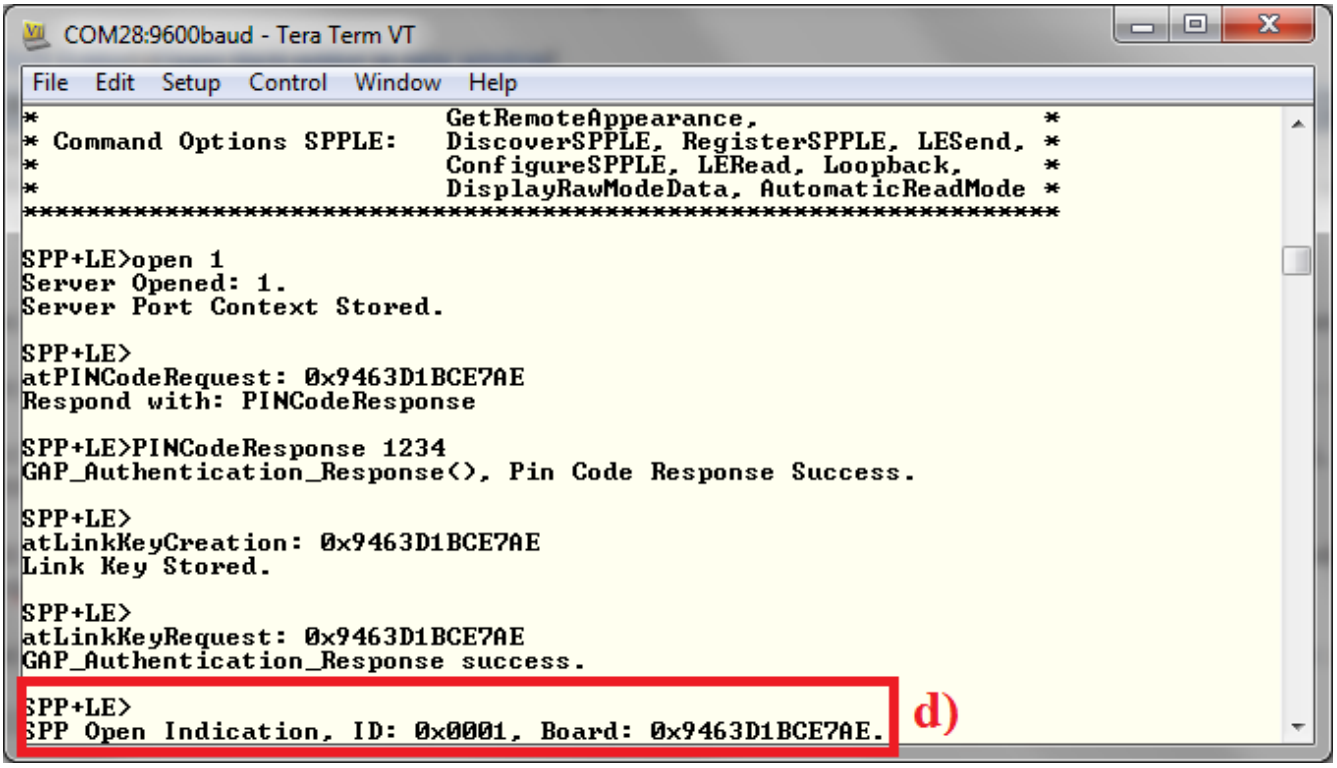

Figure 11-11. SPP Demo MSP430 PINCodeResponse

5. The two devices are now connected. Data can be sent/received from the two devices as shown here.

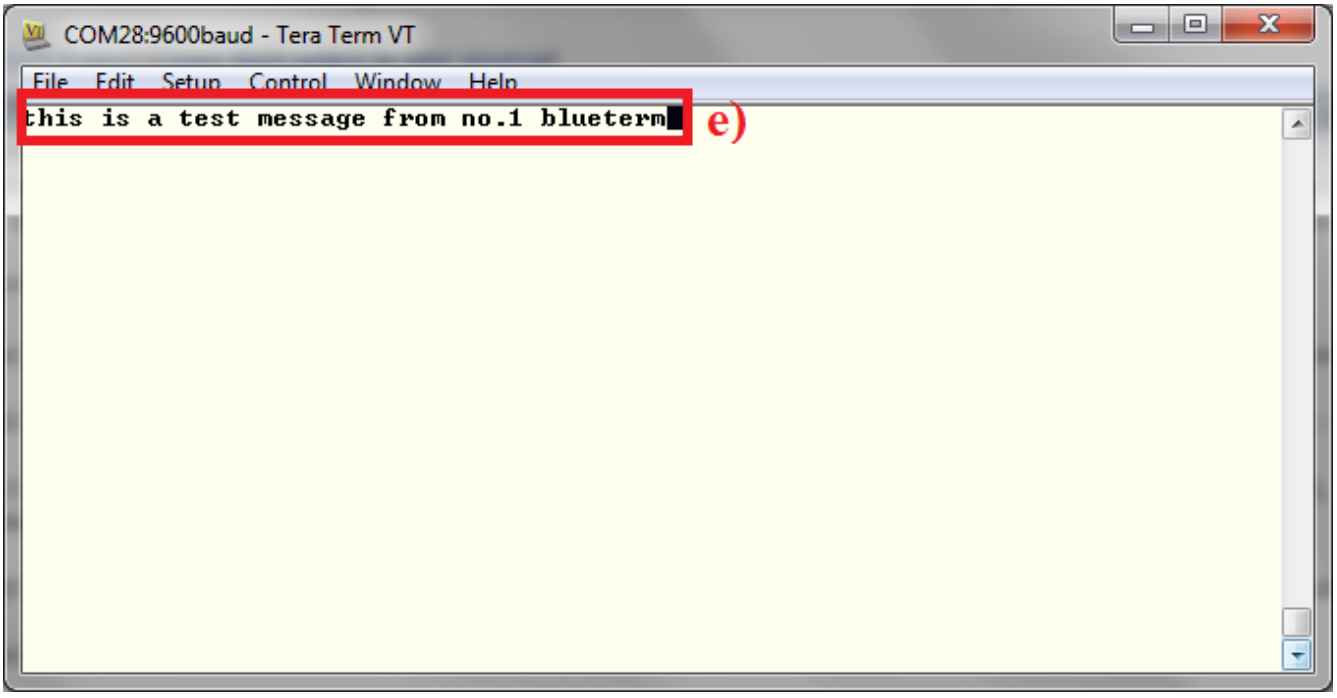

Figure 11-12. SPP Demo Test Terminal

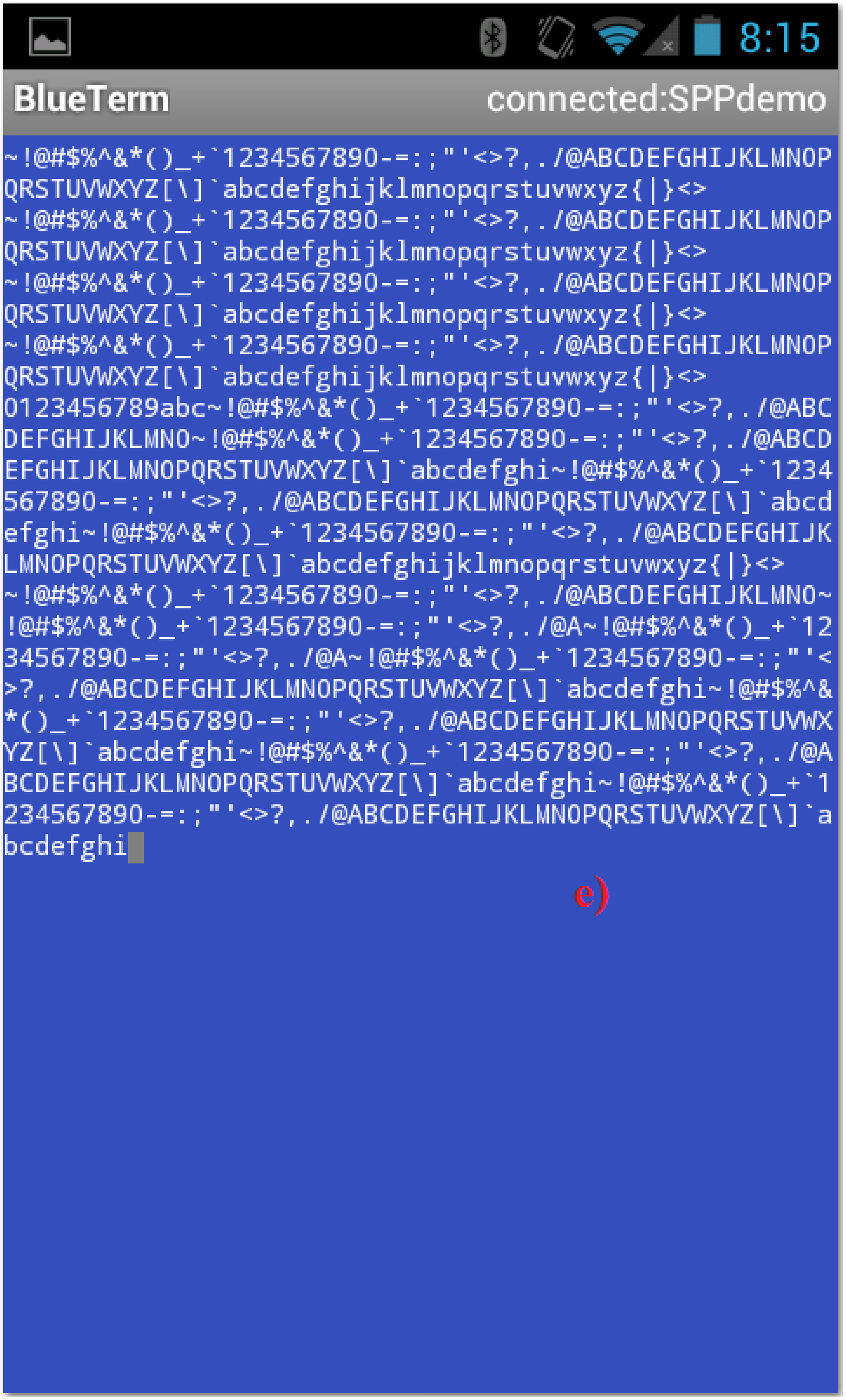

Figure 11-13. SPP Demo Test Terminal 2

Note

If you are having difficulty connecting to the sppdemo, first pair with the device under your phone's bluetooth settings and then select the device from the list of paired devices when you select connect device in blueterm. In the shipped SPP application that is a part of the SPP/SPPLE application , using the write command sends 76 bytes to the other device. It is not possible for the sender to send all 76 bytes at once because of the limitations on buffers that are smaller to reduce RAM usage. So the code sends 31 bytes and waits until the other side is ready to receive more. The other side will not be ready until it reads the sent data by calling a **read** operation. So the idea is to call Read whenever the receiver receives an SPP Data Indication and all data that was sent will be read.

An example communication:

```
SPP+LE>write

Wrote: 76.


SPP+LE>

Transmit Buffer Empty Indication, ID: 0x0001


Here is the Server side (who is receiving the data):

SPP+LE>

SPP Data Indication, ID: 0x0001, Length: 0x0026.

SPP+LE>read

Read: 31.

Message: ~!@#$%^&*()_+`1234567890-=:;"'<

Read: 7.

Message: >?,./@A

Read: 0.

SPP+LE>

SPP Data Indication, ID: 0x0001, Length: 0x0026.

SPP+LE>read

Read: 31.

Message: BCDEFGHIJKLMNOPQRSTUVWXYZ[\]`ab

Read: 7.

Message: cdefghi

Read: 0.
```

So we have:
31+31+7+7=76.

11.3 Application Commands

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands.

Serial Port Profile Commands

The Serial Port Profile defines requirements for Bluetooth Devices necessary for setting up emulated serial cable connections using RFCOMM between peer devices. One device must be the initiator and the other, the responder. This profile is built upon the Generic Access Profile and uses RFCOMM to transport user data, modem control signals, and configuration commands. A query to the Service Discovery features must be performed in order to find out the RFCOMM Server channel number of the remote device. After the query, the first step is a request for a new L2CAP channel to the remote RFCOMM entity. Then an RFCOMM session on the L2CAP channel must be initiated. A new data link connection on the RFCOMM session must be started using the server channel number found in the SDP query. After all these steps are complete, the serial cable connection is ready for use.

Read

Description

The Read command is responsible for reading data that was received via an Open SPP port. The function reads a fixed number of bytes at a time from the SPP Port and displays it. If the call to the SPP_Data_Read() function is successful but no data is available to read the function displays "No data to read." This function requires that a valid Bluetooth Stack ID and Serial Port ID exist before running. This function returns zero if successful and a negative value if an error occurred.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of Read.

Possible Return Values

(0) Successfully Read Data

(-6) INVALID_PARAMETERS_ERROR

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

(-86) BTPS_ERROR_SPP_PORT_NOT_OPENED

API Call

SPP_Data_Read(BluetoothStackID, SerialPortID, (Word_t)(sizeof(Buffer)-1), (Byte_t)&Buffer);*

API Prototype

*int BTPSAPI SPP_Data_Read(unsigned int BluetoothStackID, unsigned int SerialPortID, Word_t DataBufferSize, Byte_t *DataBuffer);*

Description of API

This function is used to read serial data from the specified serial connection. The SerialPortID that is passed to this function must have been established by either accepting a Serial Port Connection (callback from the SPP_Open_Server_Port() function) or by initiating a Serial Port Connection (via calling the SPP_Open_Remote_Port() function and having the remote side accept the connection).

Write

Description

The Write command is responsible for Writing Data to an Open SPP Port. The string that is written is defined by the constant TEST_DATA (at the top of this file). This function requires that a valid Bluetooth Stack ID and Serial Port ID exist before running. This function returns zero is successful or a negative return value if there was an error.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of Write.

Possible Return Values

(0) Successfully Wrote Data

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

(-86) BTPS_ERROR_SPP_PORT_NOT_OPENED

API Call

*SPP_Data_Write(BluetoothStackID, SerialPortID, (Word_t)BTPS_StringLength(TEST_DATA), (Byte_t *)TEST_DATA);*

API Prototype

*int BTPSAPI SPP_Data_Write(unsigned int BluetoothStackID, unsigned int SerialPortID, Word_t DataLength, Byte_t *DataBuffer);*

Description of API

This function is used to send data to the specified Serial Connection. The SerialPortID that is passed to this function must have been established by either accepting a Serial Port Connection (callback from the SPP_Open_Server_Port() function) or by initiating a Serial Port Connection (via calling the SPP_Open_Remote_Port() function and having the remote side accept the connection). If this function is unable to send all of the data that was specified (via the DataLength parameter) because of a full Transmit Buffer condition, this function will return the number of bytes that were actually sent (zero or more, but less than the DataLength parameter value). When this happens (and only when this happens), the user can expect to be notified when the Serial Port is able to send data again via the the etPort_Transmit_Buffer_Empty_Indication SPP Event. This will allow the user a mechanism to know when the Transmit Buffer is empty so that more data can be sent.

GetConfigParams

Description

The GetConfigParams command is responsible for querying the current configuration parameters that are used by SPP. This command requires that a valid Bluetooth Stack ID exists before running. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the query.

Possible Return Values

(0) Successfully Queried Configuration Parameters

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

API Call

SPP_Get_Configuration_Parameters(BluetoothStackID, &SPPConfigurationParams);

API Prototype

*int BTPSAPI SPP_Get_Configuration_Parameters(unsigned int BluetoothStackID, SPP_Configuration_Params_t *SPPConfigurationParams);*

Description of API

This function is used to determine the current SPP parameters that are being used. These parameters are the RFCOMM Frame size that is to be used for incoming/outgoing connections and the size (in bytes) of the default transmit and receive buffers that are used. The transmit and receive buffer sizes are the sizes that are used by default for newly opened SPP Ports (either client or server). The programmer is free to use the SPP_Change_Buffer_Size() function to change the transmit and receive buffer sizes for an existing SPP Port (either client or server).

SetConfigParams

Description

The SetConfigParams command is responsible for setting the current configuration parameters that are used by SPP. This function will return zero on successful execution and a negative value on errors. This command requires that a valid Bluetooth Stack ID exists before running.

Parameters

This command requires three parameters to work. The first parameter is the MaximumFrameSize, followed by the TransmitBufferSize (which can be set to 0 to keep its value), followed by the ReceiveBufferSize (which can be set to 0 to keep its value).

Command Call Examples

"SetConfigParams 0x03F9 0 0" Attempts to set the Maximum Frame Size to 1017 frames and keeps the values of Transmit Buffer Size and Receive Buffer Size to the same values that were previously set.

"SetConfigParams 0x64 200 300" Attempts to set the Maximum Frame Size to 100 frames, the Transmit Buffer Size to 200, and the Receive Buffer Size to 300.

Possible Return Values

(0) Successfully Set Configuration Parameters

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

API Call

SPP_Set_Configuration_Parameters(BluetoothStackID, &SPPConfigurationParams);

API Prototype

*int BTPSAPI SPP_Set_Configuration_Parameters(unsigned int BluetoothStackID, SPP_Configuration_Params_t *SPPConfigurationParams);*

Description of API

This function is used to change the current SPP parameters that are to be used for future SPP Ports that are opened. These parameters are the RFCOMM Frame size that is to be used for incoming/outgoing connections

and the size (in bytes) of the default transmit and receive buffers that are used. The transmit and receive buffer sizes are the sizes that are used by default for newly opened SPP Ports (either client or server). The programmer is free to use the SPP_Change_Buffer_Size() function to change the transmit and receive buffer sizes for an existing SPP Port (either client or server). This function cannot be called if there exists ANY active SPP Client of Server. In other words, these parameters can only change when there are no active SPP Server Ports or SPP Client Ports open. Note that for all of the parameters there exists special constants which indicate to use the currently configured parameters.

GetQueueParams

Description

The GetQueueParams command is responsible for querying the current queuing parameters that are used by SPP/RFCOMM (into L2CAP). This function will return zero on successful execution and a negative value on errors. This command requires that a valid Bluetooth Stack ID exists before running.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the query.

Possible Return Values

(0) Successfully Queried Queue Parameters

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

API Call

SPP_Get_Queuing_Parameters(BluetoothStackID, &MaximumNumberDataPackets, &QueuedDataPacketsThreshold);

API Prototype

*int BTPSAPI SPP_Get_Queuing_Parameters(unsigned int BluetoothStackID, unsigned int *MaximumNumberDataPackets,unsigned int *QueuedDataPacketsThreshold);*

Description of API

This function is responsible for querying the lower level data queuing parameters. These parameters are used to control the lower level data packet queuing thresholds (to improve RAM usage). Specifically, these parameters are used to control aspects of the number of data packets that can be queued into the lower level (per individual channel). This mechanism allows for the flexibility to limit the amount of RAM that is used for streaming type applications (where the remote side has a large number of credits that were granted). If both parameters are zero the queuing mechanism is disabled. This means that the number of queued packets will only be limited via the amount of available RAM. These parameters do not affect the transmit and receive buffers and do not affect any frame sizes and/or credit logic. These parameters ONLY affect the number of simultaneous data packets queued into the lower level.

SetQueueParams

Description

The SetQueueParams command is responsible for setting the current queuing parameters that are used by SPP/RFCOMM (into L2CAP). This function will return zero on successful execution and a negative value on errors. This command requires that a valid Bluetooth Stack ID exists before running.

Parameters

This command requires two parameters to work correctly. The first parameter is the Maximum Number of Data Packets. The second is the Queued Data Packets Threshold.

Command Call Examples

"SetQueueParams 100 5" Attempts to set the Maximum Number of Data Packets to 100 Packets and the Queued Data Packets Threshold to 5 Packets.

"SetQueueParams 25 1" Attempts to set the Maximum Number of Data Packets to 25 Packets and the Queued Data Packets Threshold to 1 Packets.

Possible Return Values

(0) Successfully Set Queue Parameters

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

API Call

SPP_Set_Queuing_Parameters(BluetoothStackID, (unsigned int)(TempParam->Params[0].intParam), (unsigned int)(TempParam->Params[1].intParam));

API Prototype

int BTPSAPI SPP_Set_ Queuing_Parameters(unsigned int BluetoothStackID, unsigned int MaximumNumberDataPackets, unsigned int QueuedDataPacketsThreshold);

Description of API

This function is responsible for setting the lower level data queuing parameters. These parameters are used to control the lower level data packet queuing thresholds (to improve RAM usage). Specifically, these parameters are used to control aspects of the number of data packets that can be queued into the lower level (per individual channel). This mechanism allows for the flexibility to limit the amount of RAM that is used for streaming type applications (where the remote side has a large number of credits that were granted). This function can only be called when there are NO active connections. Setting both parameters to zero will disable the queuing mechanism. This means that the number of queued packets will only be limited via the amount of available RAM. These parameters do not affect the transmit and receive buffers and do not affect any frame sizes and/or credit logic. These parameters ONLY affect the number of simultaneous data packets queued into the lower level.

Send (SendData)

Description

The SendData command is responsible for sending a number of characters to a remote device to which a connection exists. The function receives a parameter that indicates the number of byte to be transferred. This function will return zero on successful execution and a negative value on errors. There must be a connection established (with a Serial Port ID) to use this function.

Parameters

The only parameter necessary is the number of bytes to send. This value has to be greater than zero.

Command Call Examples

"Send 100" Attempts to send 100 bytes of data.

"Send 25" Attempts to send 25 bytes of data.

Possible Return Values

(0) Successfully Sent Data

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

API Call

*SPP_Data_Write(BluetoothStackID, SerialPortID, DataCount, (unsigned char *)DataStr);*

API Prototype

*int BTPSAPI SPP_Data_Write(unsigned int BluetoothStackID, unsigned int SerialPortID, Word_t DataLength, Byte_t *DataBuffer);*

Description of API

This function is used to send data to the specified Serial Connection. The SerialPortID that is passed to this function must have been established by either accepting a Serial Port Connection (callback from the SPP_Open_Server_Port() function) or by initiating a Serial Port Connection (via calling the SPP_Open_Remote_Port() function and having the remote side accept the connection). If this function is unable to send all of the data that was specified (via the DataLength parameter) because of a full Transmit Buffer condition, this function will return the number of bytes that were actually sent (zero or more, but less than the DataLength parameter value). When this happens (and only when this happens), the user can expect to be notified when the Serial Port is able to send data again via the the etPort_Transmit_Buffer_Empty_Indication SPP Event. This will allow the user a mechanism to know when the Transmit Buffer is empty so that more data can be sent.

Open (OpenRemoteServer)

Description

The Open command (when in Client Mode) is responsible for initiating a connection with a Remote Serial Port Server. This function returns zero if successful and a negative value if an error occurred. The Bluetooth Stack ID must be valid and the device must be in Client Mode. A Serial Port must not already be opened for this command to work.

Parameters

The command takes two parameters to work. The first is the Inquiry Index which can be found using the DisplayInquiryList command after an Inquiry has been completed. The second is the RFCOMM Server Port which will be used to open a Remote SPP Port.

Command Call Examples

"Open 12 4" Attempts to Open a Remote Serial Port Server with the Remote Bluetooth Device whose address is found at the twelfth Inquiry Index using RFCOMM Server Port #4.

"Open 1 1" Attempts to Open a Remote Serial Port Server with the Remote Bluetooth Device whose address is found at the first Inquiry Index using RFCOMM Server Port #1.

"Open 19 3" Attempts to Open a Remote Serial Port Server with the Remote Bluetooth Device whose address is found at the nineteenth Inquiry Index using RFCOMM Server Port #3.

Possible Return Values

(0) Successfully Opened Remote Server

(-8) INVALID_STACK_ID_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

(-86) BTPS_ERROR_SPP_PORT_NOT_OPENED

(-72) BTPS_ERROR_RFCOMM_UNABLE_TO_CONNECT_TO_REMOTE_DEVICE

(-73) BTPS_ERROR_RFCOMM_UNABLE_TO_COMMUNICATE_WITH_REMOTE_DEVICE

API Call

SPP_Open_Remote_Port(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam - 1)], TempParam->Params[1].intParam, SPP_Event_Callback, (unsigned long)0);

API Prototype

int BTPSAPI SPP_Open_Remote_Port(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, unsigned int ServerPort, SPP_Event_Callback_t SPP_Event_Callback, unsigned long CallbackParameter);

Description of API

This function is used to open a remote serial port on the specified Remote Device.

Close (CloseRemoteServer)

Description

The Close command (when in Client Mode) is responsible for terminating a connection with a Remote Serial Port Server. This function returns zero if successful and a negative value if an error occurred. The Bluetooth Stack ID must be valid, the device must be in Client Mode, and a Remote Serial Port Server must exist for the command to work.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the server closing.

Possible Return Values

(0) Successfully Closed Server

(-8) INVALID_STACK_ID_ERROR

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

(-86) BTPS_ERROR_SPP_PORT_NOT_OPENED

API Call

SPP_Close_Port(BluetoothStackID, SerialPortID);

API Prototype

int BTPSAPI SPP_Close_Port(unsigned int BluetoothStackID, unsigned int SerialPortID);

Description of API

This function is used to close a Serial Port that was previously opened with the SPP_Open_Server_Port( ) function or the SPP_Open_Remote_Port( ) function. This function does not unregister a SPP Server Port from the system; it only disconnects any connection that is currently active on the Server Port. The SPP_Close_Server_Port() function can be used to Unregister the SPP Server Port.

Open (OpenServer)

Description

The Open command (when in Server Mode) is responsible for opening a Serial Port Server on the Local Device. This function opens the Serial Port Server on the specified RFCOMM Channel. This function returns zero if successful, or a negative return value if an error occurred. The Bluetooth Stack ID must be valid, the device must be in Server Mode, and a Serial Port Server must not already exist for the command to work.

Parameters

The command only requires one parameter. This parameter is the Port Number.

Command Call Examples

"Open 4" Attempts to Open a Serial Port Server at Port Number #4.

"Open 1" Attempts to Open a Serial Port Server at Port Number #1.

Possible Return Values

(0) Successfully Opened Server

(-9) UNABLE_TO_REGISTER_SERVER

(-8) INVALID_STACK_ID_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

(-86) BTPS_ERROR_SPP_PORT_NOT_OPENED

API Calls

SPP_Open_Server_Port(BluetoothStackID, TempParam->Params[0].intParam, SPP_Event_Callback, (unsigned long)0);

SPP_Register_SDP_Record(BluetoothStackID, ServerPortID, NULL, ServiceName, &SPPServerSDPHandle);

API Prototypes

int BTPSAPI SPP_Open_Server_Port(unsigned int BluetoothStackID, unsigned int ServerPort, SPP_Event_Callback_t SPP_Event_Callback, unsigned long CallbackParameter);

*int BTPSAPI SPP_Register_SDP_Record(unsigned int BluetoothStackID, unsigned int SerialPortID, SPP_SDP_Service_Record_t *SDPServiceRecord, char *ServiceName, DWord_t *SDPServiceRecordHandle)*

Description of APIs

This function is responsible for establishing a Serial Port Server which will wait for a connection to occur on the port established by this function.

This function provides a means to add a generic SDP Service Record to the SDP Database. This function should only be called with the SerialPortID that was returned from the SPP_Open_Server_Port( ) function. This function should never be used with the Serial Port ID returned from the SPP_Open_Remote_Port( ) function. The Service Record Handle that is returned from this function will remain in the SDP Record Database until it is deleted by calling the SDP_Delete_Service_Record( ) function. A Macro is provided to delete the Service Record from the SDP Database. This Macro maps SPP_Un_Register_SDP_Record( ) to SDP_Delete_Service_Record(), and is defined as follows:

Close (CloseServer)

Description

The Close command (when in Server Mode) is responsible for closing a Serial Port Server that was previously opened via a successful call to the OpenServer() function. This function returns zero if successful or a negative return error code if there was an error. The Bluetooth Stack ID must be valid, the device must be in Server Mode, and a Serial Port Server must exist for the command to work.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the server closing.

Possible Return Values

(0) Successfully Closed Server

(-8) INVALID_STACK_ID_ERROR

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

(-86) BTPS_ERROR_SPP_PORT_NOT_OPENED

API Calls

SPP_Close_Server_Port(BluetoothStackID, ServerPortID);

SPP_Un_Register_SDP_Record(BluetoothStackID, SerialPortID, SPPServerSDPHandle);

API Prototypes

int BTPSAPI SPP_Close_Server_Port(unsigned int BluetoothStackID, unsigned int SerialPortID)

*int BTPSAPI SPP_Un_Register_SDP_Record(unsigned int BluetoothStackID, unsigned int SerialPortID, DWord_t *SDPServiceRecordHandle);*

Description of APIs

This function is responsible for Unregistering a Serial Port Server which was registered by a successful call to the SPP_Open_Server_Port( ) function. Note, this function does NOT delete any SDP Service Record Handles (i.e., added via a SPP_Register_SDP_Record( ) function call).

This function provides a means to add a generic SDP Service Record to the SDP Database. This function should only be called with the SerialPortID that was returned from the SPP_Open_Server_Port( ) function. This function should never be used with the Serial Port ID returned from the SPP_Open_Remote_Port( ) function. The Service Record Handle that is returned from this function will remain in the SDP Record Database until it is deleted by calling the SDP_Delete_Service_Record( ) function. A Macro is provided to delete the Service Record from the SDP Database. This Macro maps SPP_Un_Register_SDP_Record( ) to SDP_Delete_Service_Record(), and is defined as follows: SPP_Un_Register_SDP_Record(__BluetoothStackID, __SerialPortID, __SDPRecordHandle) If no UUID information is specified in the SDPServiceRecord Parameter, then the default SPP Service Classes are added. Any Protocol Information that is specified (if any) will be added in the Protocol Attribute after the default SPP Protocol List (L2CAP and RFCOMM). The Service Name is always added at Attribute ID 0x0100. A Language Base Attribute ID List is created that specifies that 0x0100 is UTF-8 Encoded, English Language.

Host Controller Interface Commands

The Host Controller Interface provides a uniform interface method of accessing a Bluetooth Controller's capabilities. The Host Controller driver should be independent of the underlying transport technology. The transport should not require understanding of the data that the Host Controller driver passes to the Controller. This allows for transparency in the transport layer. HCI is used for many different commands such as flow control from Host to Controller, local device information discovery, and changing global configuration parameters.

SniffMode

Description

The SniffMode command is responsible for putting a specified connection into HCI Sniff Mode with passed in parameters. There must be an SPP Connection created so that a Connection Handle exists. The command requires that a valid Bluetooth Stack ID exists before running.

Parameters

There has to be four parameters for this function to work. The first Maximum Sniff Interval, which is followed by the Minimum Sniff Interval. The third parameter is the Sniff Attempt which is followed by the Sniff Timeout. All of these parameters must be values between 0x0001 to 0xffff. The values are number of baseband slots (0.625 msec).

Command Call Examples

"SniffMode 0xffff 0x55ff 0x0fff 0x1fff" Attempts to set the connection into HCI Sniff Mode with a Maximum Sniff Interval of 40.9 seconds, a Minimum Sniff Interval of 13.8 seconds, a Sniff Attempt every 2.6 seconds, and the Sniff Timeout of 5.12 seconds.

"SniffMode 0x1111 0x0001 0x0005 0x0120" Attempts to set the connection into HCI Sniff Mode with a Maximum Sniff Interval of 2.7 seconds, a Minimum Sniff Interval of .625 milliseconds, a Sniff Attempt every 3.125 milliseconds, and the Sniff Timeout of 180 milliseconds.

Possible Return Values

(0) Successfully Entered Sniff Mode

(-6) INVALID_PARAMETERS_ERROR

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1) BTPS_ERROR_INVALID_PARAMETER

(-66) BTPS_ERROR_INSUFFICIENT_RESOURCES

(-14) BTPS_ERROR_HCI_DRIVER_ERROR

(-57) BTPS_ERROR_HCI_RESPONSE_ERROR

API Call

HCI_Sniff_Mode(BluetoothStackID, Connection_Handle, Sniff_Max_Interval, Sniff_Min_Interval, Sniff_Attempt, Sniff_Timeout, &Status);

API Prototype

*int BTPSAPI HCI_Sniff_Mode(unsigned int BluetoothStackID, Word_t Connection_Handle, Word_t Sniff_Max_Interval, Word_t Sniff_Min_Interval, Word_t Sniff_Attempt, Word_t Sniff_Timeout, Byte_t *StatusResult);*

Description of API

This command places the specified connection into Sniff Mode as per the specified parameters.

ExitSniffMode

Description

The ExitSniffMode command is responsible for exiting a specified connection that is in HCI Sniff Mode. The device must already be in Sniff Mode to correctly exit. There must be an SPP Connection created so that a Connection Handle exists. The command requires that a valid Bluetooth Stack ID exists before running.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of exiting Sniff Mode.

Possible Return Values

(0) Successfully exit Sniff Mode

(-6) INVALID_PARAMETERS_ERROR

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1) BTPS_ERROR_INVALID_PARAMETER

(-66) BTPS_ERROR_INSUFFICIENT_RESOURCES

(-14) BTPS_ERROR_HCI_DRIVER_ERROR

(-57) BTPS_ERROR_HCI_RESPONSE_ERROR

API Call

HCI_Exit_Sniff_Mode(BluetoothStackID, Connection_Handle, &Status);

API Prototype

*int BTPSAPI HCI_Exit_Sniff_Mode(unsigned int BluetoothStackID, Word_t Connection_Handle, Byte_t *StatusResult);*

Description of API

This command terminates the Sniff Mode for a connection.

SetBaudrate

Description

The SetBaudRate command is responsible for changing the current Baud Rate used to talk to the Radio. This function ONLY configures the Baud Rate for a TI Bluetooth chipset. This command requires that a valid Bluetooth Stack ID exists.

Parameters

This command requires one parameter. The value is an integer representing a value used for the Baud Rate. The options are 0 (for Baud Rate of 115200), 1 (for Baud Rate 230400), 2 (for Baud Rate 460800), 3 (for Baud Rate 921600), 4 (for Baud Rate 1843200), or 5 (for Baud Rate 3686400). The maximum baud rate default is 921600 so options 4 and 5 are disable.

Command Call Examples

"SetBaudRate 0" Attempts to set the Baud Rate to 115200.

"SetBaudRate 1" Attempts to set the Baud Rate to 230400.

"SetBaudRate 2" Attempts to set the Baud Rate to 460800.

"SetBaudRate 3" Attempts to set the Baud Rate to 921600.

Possible Return Values

(0) Successfully Set Baud Rate

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

API Call

HCI_Reconfigure_Driver(BluetoothStackID, FALSE, &(Data.DriverReconfigureData));

API Prototype

*int BTPSAPI HCI_Reconfigure_Driver(unsigned int BluetoothStackID,Boolean_t ResetStateMachines,HCI_Driver_Reconfigure_Data_t *DriverReconfigureData);*

Description of API

This function issues the appropriate call to an HCI driver to request the HCI Driver to reconfigure itself with the corresponding configuration information.

Application Specific Commands
DesplayInquiryList

Description

The DisplayInquiryList command exists to display the current Inquiry List with indexes. This command is useful for when a user has forgotten the Inquiry Index for a particular Bluetooth Device the user may want to interact with. This function returns zero on a successful execution and a negative value on all errors. The command requires that a valid Bluetooth Stack ID exists before running and it should be called after using the Inquiry command, since the list would be empty without already discovering devices.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Inquiry List displayed.

Possible Return Values

(0) Successful Display of the Inquiry List

(-8) INVALID_STACK_ID_ERROR

ChangeSimplePairingParameters

Description

The ChangeSimplePairingParameters command is responsible for changing the Secure Simple Pairing Parameters that are exchanged during the Pairing procedure whenSecure Simple Pairing (Security Level 4) is used. This function returns zero on a successful execution and a negative value on all errors. A Bluetooth Stack ID must exist before attempting to call this function. The IOCapability and MITMProtection values are stored in static global variables which are used for Secure Simple Pairing.

Parameters

This command requires two parameters which are the I/O Capability and the MITM Requirement. The first parameter must be specified as 0 (for Display Only), 1 (for Display Yes/No), 2 (for Keyboard Only), or 3 (for No Input/Output). The second parameter must be specified as 0 (for No MITM) or 1 (for MITM required).

Command Call Examples

"ChangeSimplePairingParameters 3 0" Attempts to set the I/O Capability to No Input/Output and turns off MITM Protection.

"ChangeSimplePairingParameters 2 1" Attempts to set the I/O Capability to Keyboard Only and activates MITM Protection.

"ChangeSimplePairingParameters 1 1" Attempts to set the I/O Capability to Display Yes/No and activates MITM Protection.

Possible Return Values

(0) Successfully Pairing Parameters Change

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

Loopback

Description

The Loopback command is responsible for setting the application state to support loopback mode. This function will return zero on successful execution and a negative value on errors. This command requires that a valid Bluetooth Stack ID exists before running.

Parameters

There is only one parameter for this command. If the first parameter value is 0, Loopback is turned off. Any other value would set Loopback as active. If no parameter is given, the setting will be turned off.

Command Call Examples

"Loopback 0" Attempts to turn the Loopback off.

"Loopback" This is the same as the above example.

"Loopback 1" Attempts set Loopback as active.

"Loopback 124 512" Also sets the Loopback as active. The values are not important as long as there is a parameter and the value is not "0."

Possible Return Values

(0) Successfully Set Loopback Mode

(-6) INVALID_PARAMETERS_ERROR

DisplayRawModeData

Description

The DisplayRawData command is responsible for setting the application state to support displaying Raw Data. This function will return zero on successful execution and a negative value on errors. This command requires that a valid Bluetooth Stack ID exists and Loopback is inactive before running.

Parameters

There is only one parameter for this command. If the first parameter value is 0, Display Raw Data is turned off. Any other value would set Display Raw Data as active. If no parameter is given, the setting will be turned off.

Command Call Examples

"DisplayRawModeData 0" Attempts to turn the Display Raw Mode Data off.

"DisplayRawModeData" This is the same as the above example.

"DisplayRawModeData 1" Attempts set Display Raw Mode Data as active.

"DisplayRawModeData 124 512" Also sets the Display Raw Mode Data as active. The values are not important as long as there is a parameter and the value is not "0."

Possible Return Values

(0) Successfully Set Display Raw Data

(-6) INVALID_PARAMETERS_ERROR

AutomaticreadMode

Description

The AutomaticReadMode command is responsible for setting the application state to support Automatically reading all data that is received through SPP. This function will return zero on successful execution and a negative value on errors. This command requires that a valid Bluetooth Stack ID exists and Loopback is inactive before running.

Parameters

There is only one parameter for this command. If the first parameter value is 0, Automatic Read Mode is turned off. Any other value would set Automatic Read Mode as active. If no parameter is given, the setting will be turned off.

Command Call Examples

"AutomaticReadMode 0" Attempts to turn Automatic Read Mode off.

"AutomaticReadMode" This is the same as the above example.

"AutomaticReadMode 1" Attempts set Automatic Read Mode as active.

"AutomaticReadMode 124 512" Also sets Automatic Read Mode as active. The values are not important as long as there is a parameter and the value is not "0."

Possible Return Values

(0) Successfully Set Automatic Read Mode

(-6) INVALID_PARAMETERS_ERROR

Help (DisplayHelp)

Description

The DisplayHelp command will display the Command Options menu. Depending on the UI_MODE of the device (Server or Client), different commands will be used in certain situations. The Open and Close commands change their use depending on the mode the device is in.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Help Menu.

Possible Return Values

The return value is always 0

MemoryUsuage (QueryMemory)

Description

The MemoryUsage command is responsible for querying the memory usage. This function will return zero.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

The return value is always 0

12 SPPLE Demo Guide

12.1 Demo Overview

Note

The same instructions can be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms.

This application demonstrates a BR/EDR SPP based application as well as a custom application, SPPLE, over Bluetooth LE that is similar in functionality to the BR/EDR appliacation. The SPPLE Profile is similar to the SPP profile except that it uses LE transport compared to BR/EDR transport in the SPP profile.

The SPP profile emulates serial cable connections. There are two roles defined in this profile. The first is the server that has the SPPLE service running on it and has open an server port. The client is a device that connects to the server. Both of these devices can then exchange data with each other.

This document talks about the SPPLE application in details.

To read more about the BR/EDR version of SPP refer to this document SPP profile.

The application allows the user to use a console to use Bluetooth Low Energy (BLE) to establish connection between two BLE devices, send Bluetooth commands and exchange data over BLE.

It is recommended that the user visits the kit setup Getting Started Guide for MSP430, Getting Started Guide for TIVA, Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as **MSP-EXP430F5438 USB - Serial Port(COM x)**, **Tiva Virtual COM Port (COM x)**, **XDS110 Class Application/User UART (COM x)** for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud (9600 for MSP430), 8, n, 1. Once connected, reset the device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands. This device will become the server.

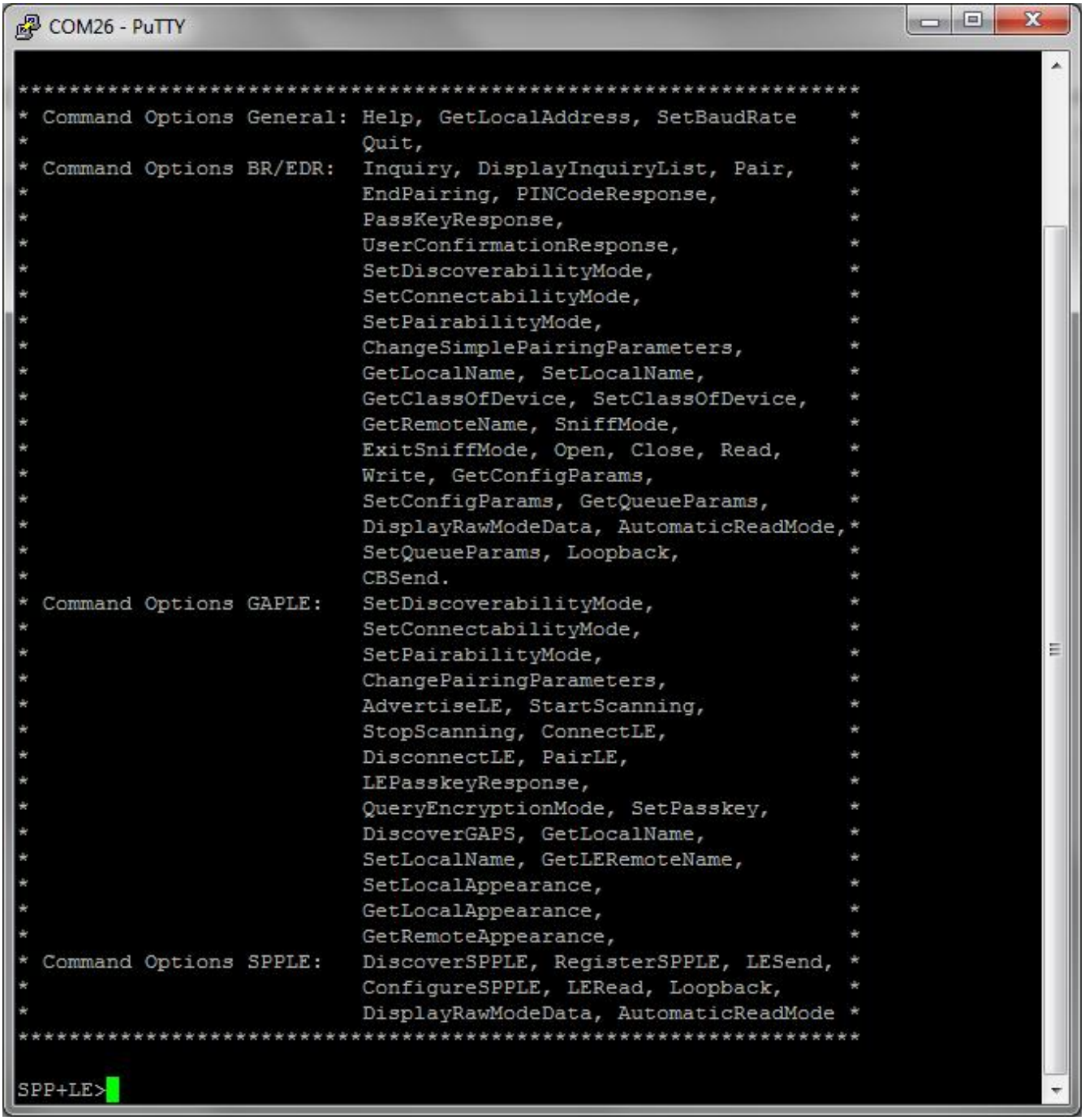

Figure 12-1. SPPLE Demo LE Terminal

Now connect the second board via miniUSB or microUSB cable and follow the same steps performed before when running the Bluetooth code on the first board. The second device that is connected to the computer will be the client.

12.2 Demo Application

Below is a description on how to use the demo application to connect two configured boards and communicate over bluetoothLE. The included application registers a custom service on a board when the stack is initialized.

Device 1 (Server) setup on the demo application

1. To start, place the device into server mode by typing: **Server** on the console. The SPP-LE Service can then be started by running **RegisterSPPLE**.

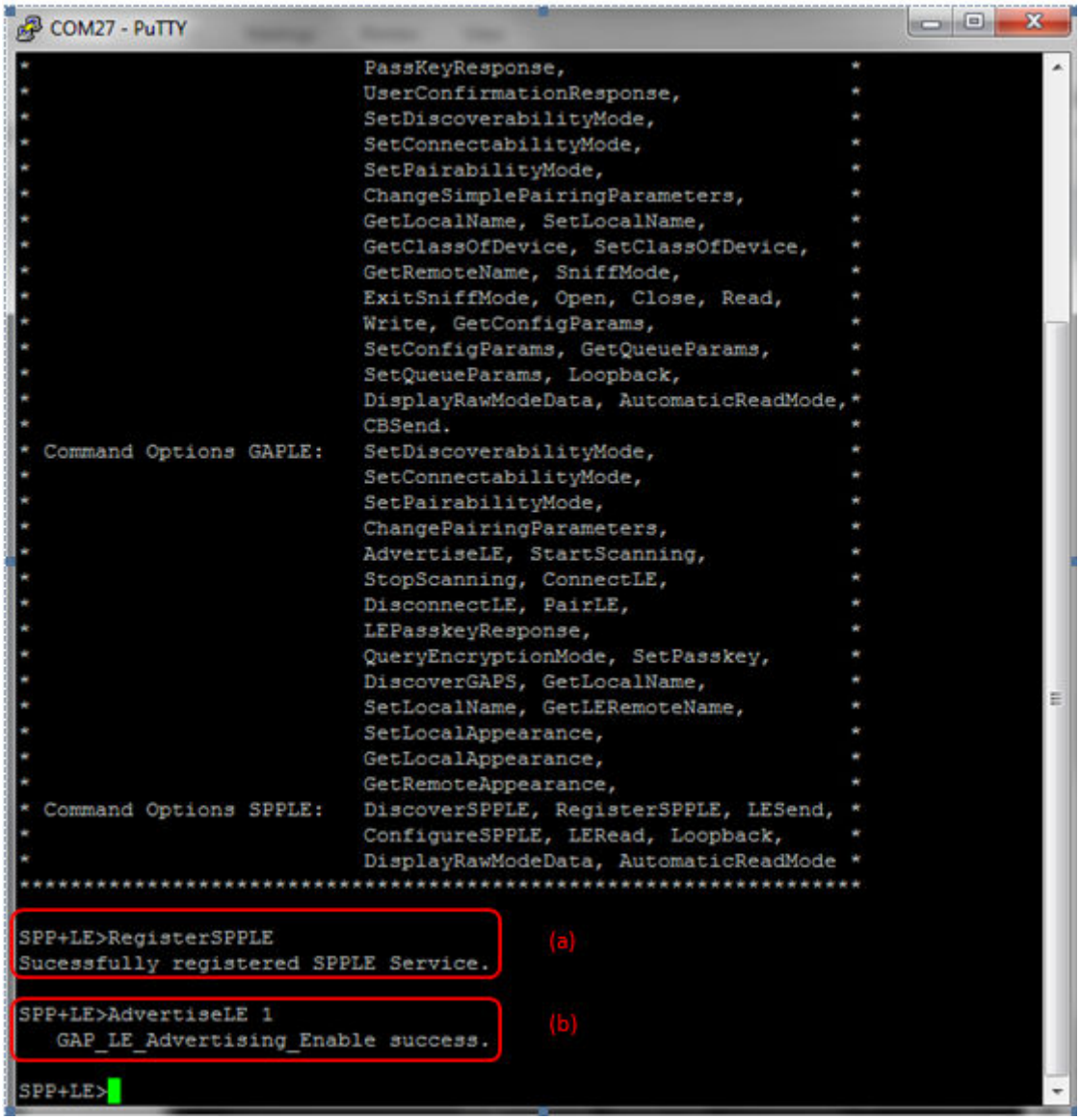

Figure 12-2. SPP Demo LE Terminal 2

2. Next, the device acting as a server needs to advertise to other devices. This can be done by running **AdvertiseLE 1**.

Device 2 (Client) setup on the demo application

1. Place the device into client mode by typing **Client** on the console.[Steps d and e are optional if you already know the Bluetooth address of the device that you want to connect to]
2. The client LE device can try to find which LE devices are in the vicinity using the command: **StartScanning**.
3. Once you have found the device, you can stop scanning by using the command: **StopScanning**.

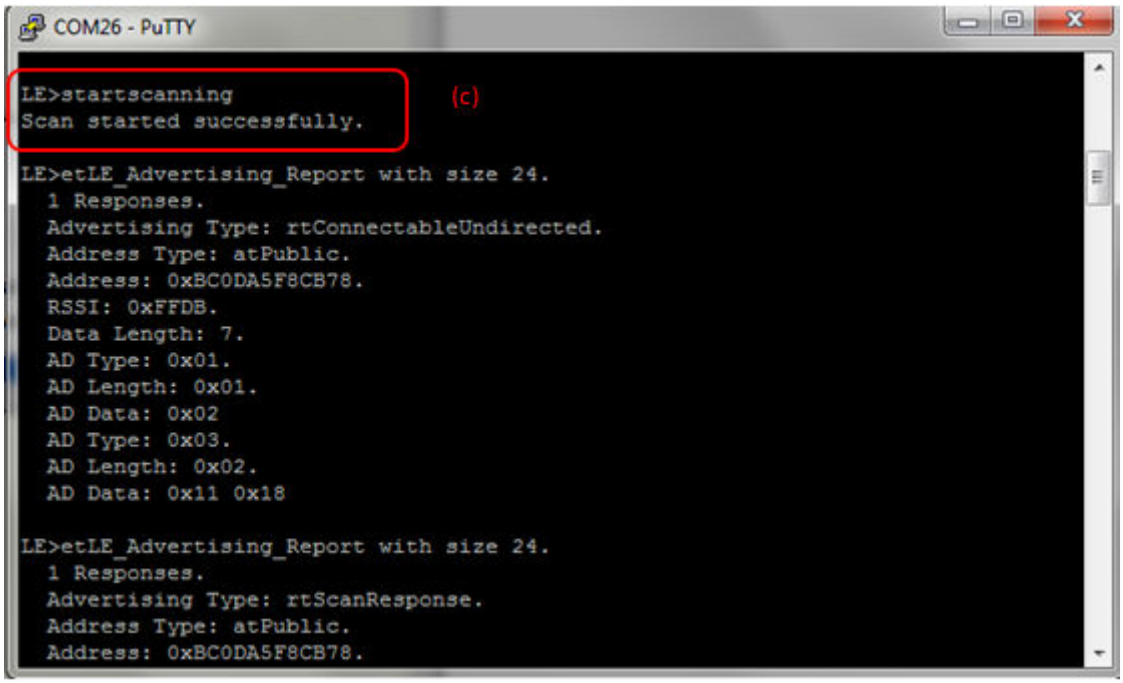

Figure 12-3. SPPLE Demo LE Terminal 3

Initiating connection from device 2

1. Once the application on the client side knows the Bluetooth address of the device that is advertising, it can connect to that device using the command: **ConnectLE <Bluetooth Address>**

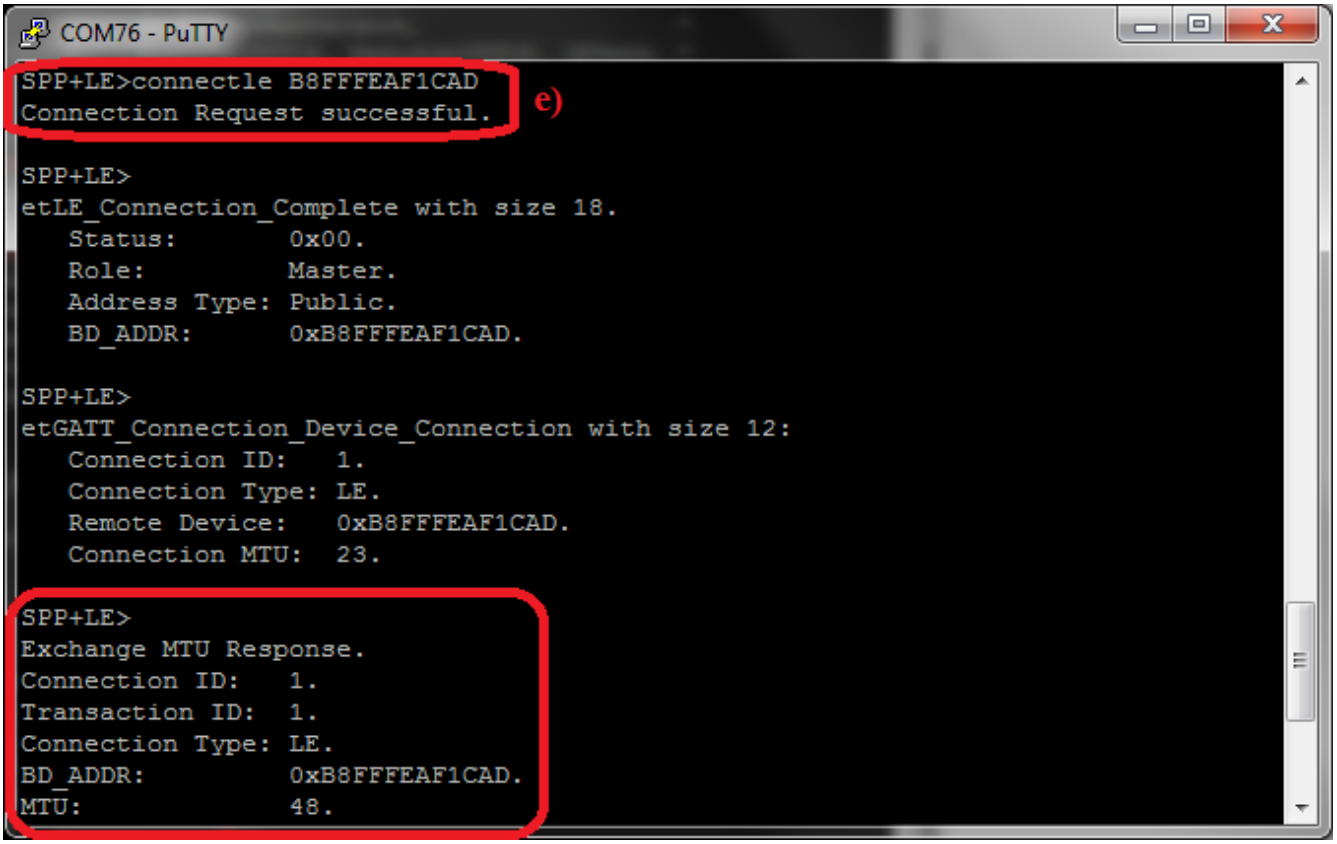

Figure 12-4. SPPLE Demo LE Terminal 4

Identify supported services

1. After Initialization, the device needs to find out if SPP services are supported. To do this run **DiscoverSPPLE <Server BD-Address>** on the client.

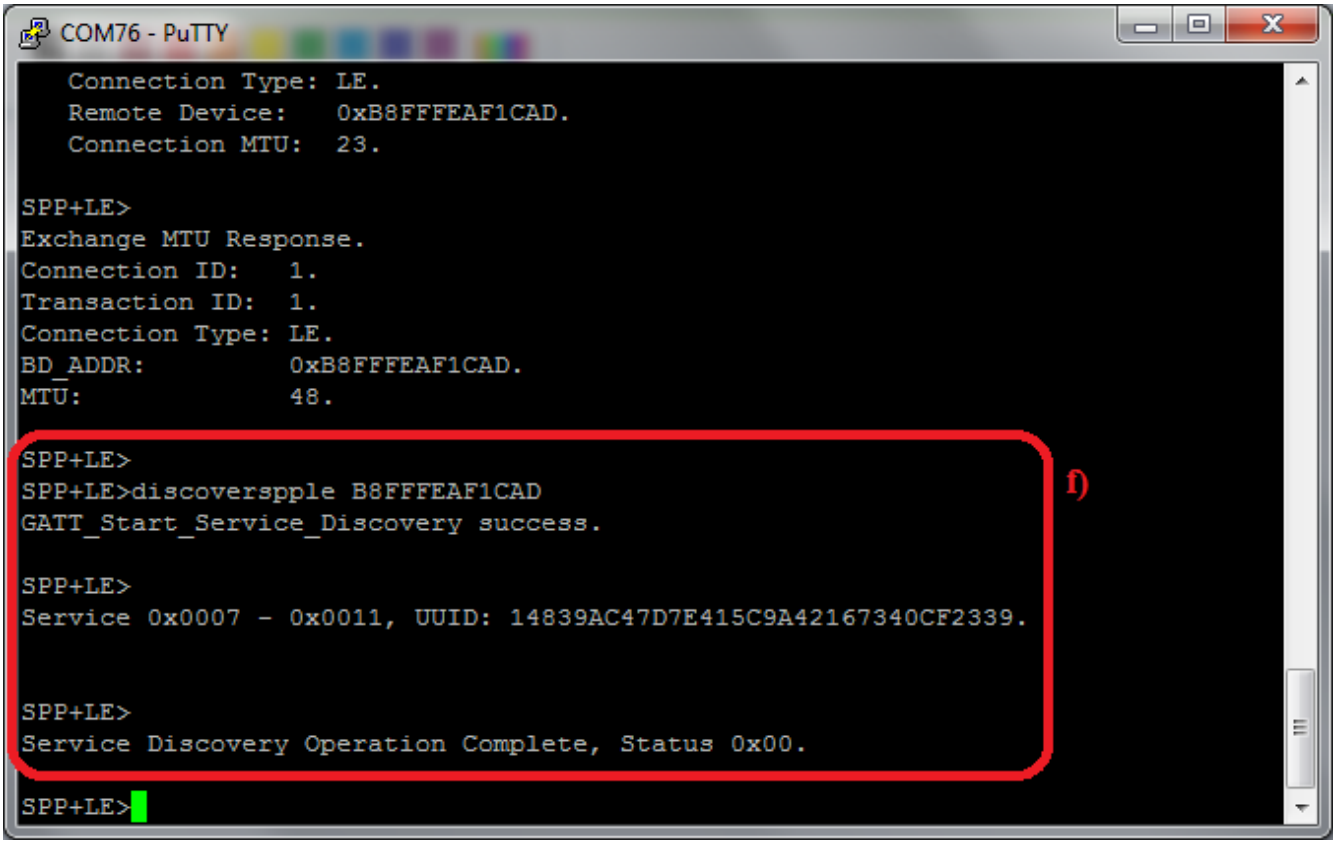

Figure 12-5. SPPLE Demo LE Terminal 5

2. After finding out support for SPP-LE, we need to configure SPP-LE. This is done by running **ConfigureSPPLE <Server BD-Address>** on the client.

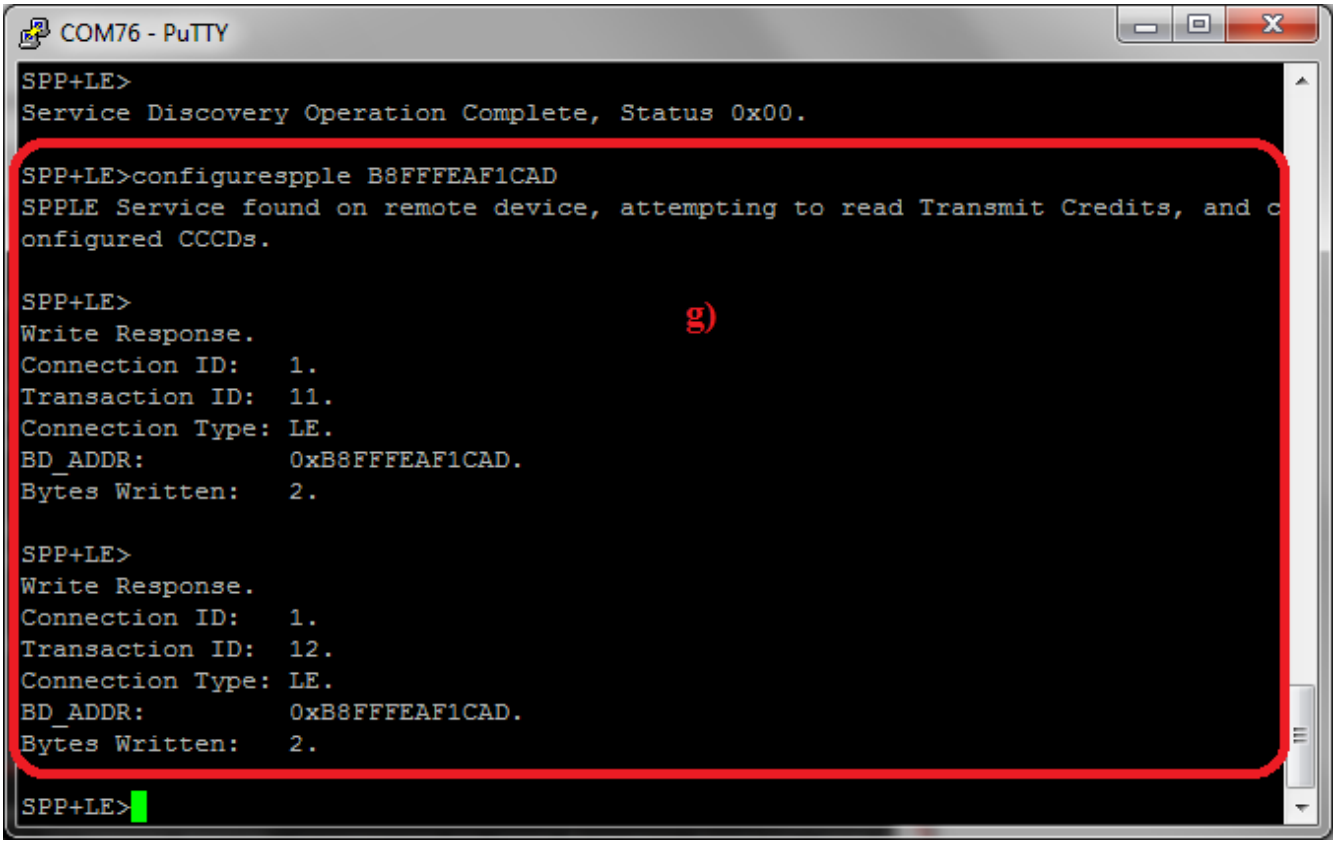

Figure 12-6. SPPLE Demo LE Terminal 6

Data Transfer between Client and Server

1. After configuring we can send data between client and server. To send data we use **LESend <Remote Device BD-Address> <Number of bytes>**.

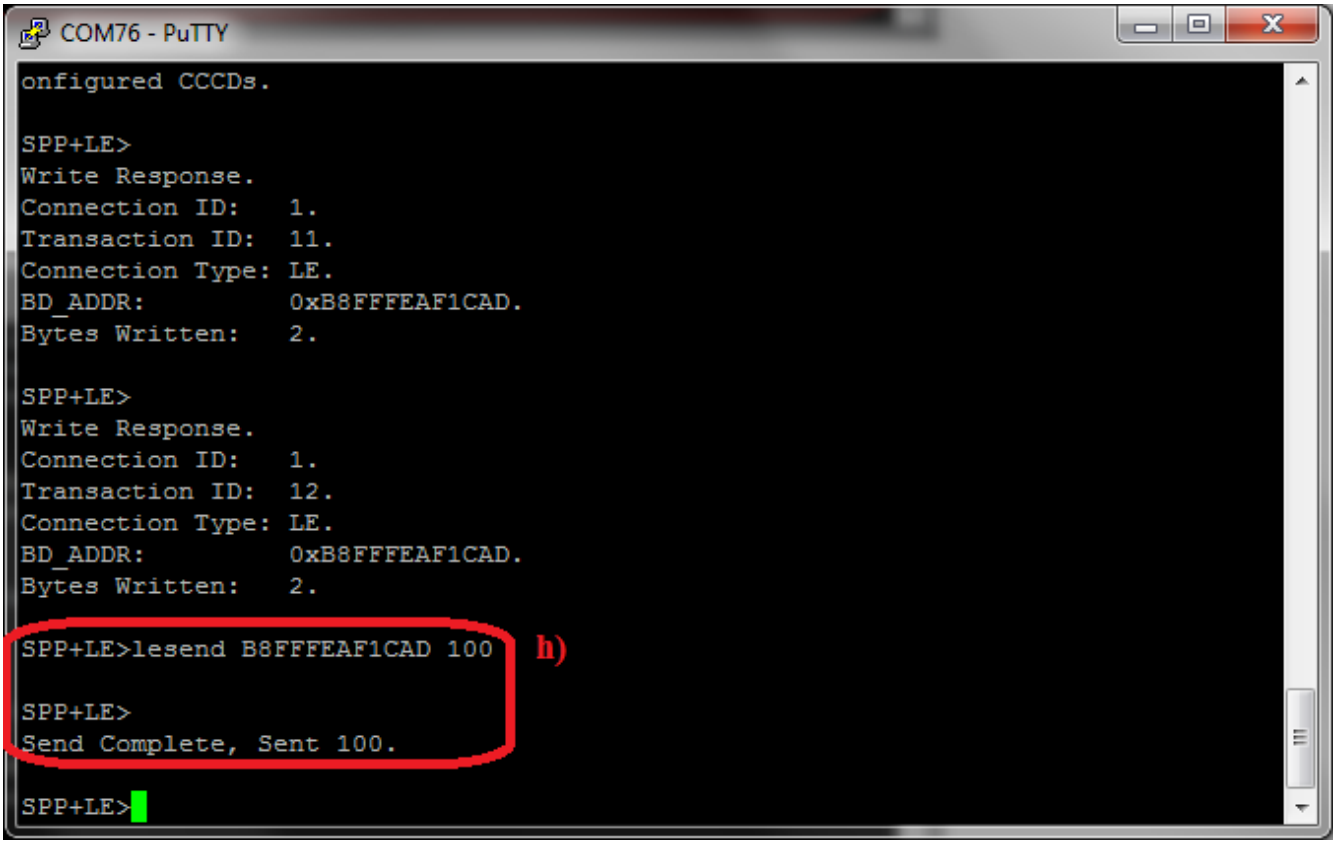

Figure 12-7. SPPLE Demo LE Terminal 7

2. Once the other device receives the data it receives a Data Indication event.
3. The receiving device can then read the data that was sent using command: **LERead <Remote Device BD-Address>**.

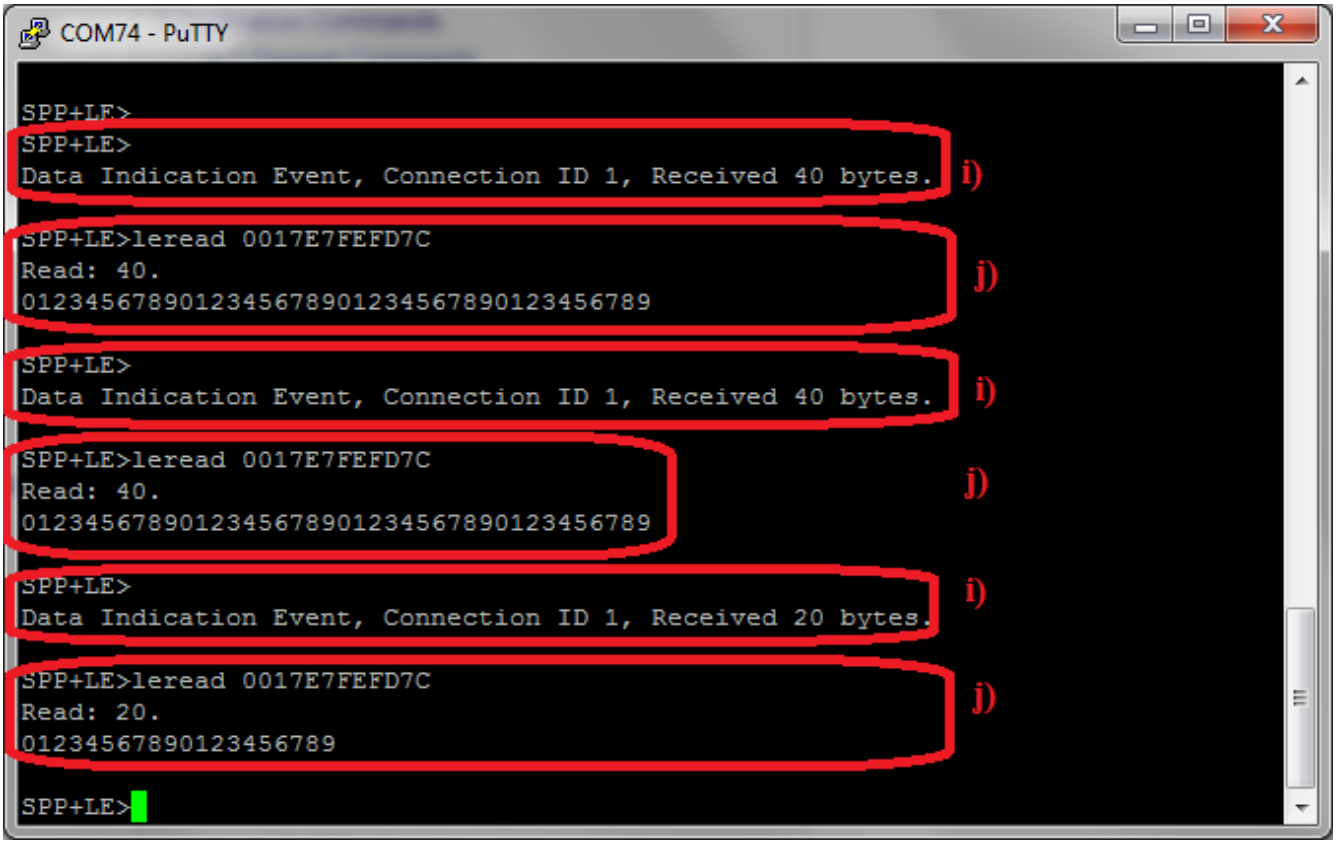

Figure 12-8. SPPLE Demo LE Terminal 8

4. This will print out the data that was sent. This data was sent over BluetoothLE using a custom service of SPPLE in the sample application.

Multiple SPPLE Connections Guide

Two SPPLE Connections

1. In This version, we test two simultaneous SPPLE connections to the MSP430. The remote devices are used as a peripheral device while the MSP430 acts as the central device.
2. Connect to the first device, discover and configure services on the first device. When discovering services and configuring services we have to specify the remote BD_ADDR that we connected to.
3. Similarly, Connect to the second device, discover and configure services on the second device.
4. To send data to the first remote device data we use **LeSend <BD-ADDR> <Number of Bytes to be sent>**
5. To send data to the second remote device data we use **LeSend <BD-ADDR> <Number of Bytes to be sent>**
6. To read data from the first remote device data we use **LeRead <BD-ADDR>**
7. To read data from the second remote device data we use **LeRead <BD-ADDR>**
8. When we turn on **Automaticreadmode, DisplayRawmodedata** or **Loopback** it turns it on for both connections.

One SPP and One SPPLE Connection

1. In this version, we test an SPP connection and SPPLE Connection at the same time to the MSP430. One of the remote devices is used as a peripheral LE device while the remote device as SPP Client.
2. Connect to the first device, discover and configure services on the first device. When discovering services and configuring services we have to specify the remote BD_ADDR that we connected to.
3. Open an SPP server and let the second remote device connect to it.
4. To send data to the first remote device data we use **LeSend <BD-ADDR> <Number of Bytes to be sent>**
5. To send data to the second remote data we use **CBSend <Number of Bytes to be sent> <Serial Port ID>**. If we want to write a small amount of data we use the command **Write <Serial Port ID>**
6. To read data from the first remote device data we use **LeRead <BD-ADDR>**
7. To read data from the second remote device data we use **Read**.

8. When we turn on **Automaticreadmode, DisplayRawmodedata** or Loopback it turns it on for both connections.

12.3 Demonstrating SPP LE on an iOS Device with the LightBlue App

LightBlue Overview

The LightBlue app is a free iOS app that allows you to test and demonstrate the GATT Profile using Bluetooth Low Energy (BLE). It allows you to create custom services and interact with servers with custom services. The app supports both the client and server roles of GATT. Here we will explain how to use the app with the SPPLEDemo application.

SPP LE Service Overview

SPP LE is not an official Bluetooth service. It is a custom service that is designed to demonstrate using Bluetooth Low Energy to send and receive data in a similar manner that Classic Bluetooth's SPP profile does. It uses a credit based protocol to send and receive data. In order for a device to send data to a remote device with the SPP LE protocol the remote device must have provided the device with "credits". These credits specify how much data the device is allowed to send. When a device has sent its maximum number of credits, it must wait for the remote device to provide it with more credits before it can continue sending. In this application 1 credit is equivalent to 1 byte (octet) of data.

Characteristics

SPP LE implements its credit-based protocol using GATT characteristics. The SPP LE service has 4 characteristics:

Name	UUID	Purpose
Rx Characteristic	0x8B00ACE7-EB0B-49B0-BBE9-9AEE0A26E1A3	Client sends data to the server using this characteristic with an ATT Write Request.
Tx Credits Characteristic	0xBA04C4B2-892B-43BE-B69C-5D13F2195392	Client sends its credits to the server using this characteristic with an ATT Write Request.
Tx Characteristic	0x0734594A-A8E7-4B1A-A6B1-CD5243059A57	Server sends data to the client using this characteristic with an ATT Handle Value Notification.
Rx Credits Characteristic	0xE06D5EFB-4F4A-45C0-9EB1-371AE5A14AD4	Server sends its credits to the client using this characteristic with an ATT Handle Value Notification.

The client and server use these characteristics to send and receive data and credits. Next we'll demonstrate SPPLEDemo as the server and LightBlue as the client. If you haven't already done so, download the LightBlue app from the App Store and turn on Bluetooth on your iOS device.

Note

For more information about characteristics, ATT Write Requests, and ATT Handle Value Notifications, please refer to the Attribute Protocol (ATT) and Generic Attribute Profile (GATT) specifications in the Bluetooth Core specification, which can be found on the Bluetooth SIG's website. The following instructions were confirmed in version 2.2.0 of LightBlue running on an iPhone 5 with iOS 8.1.3. These instructions can be used with the SPPLEDemo app from any TI Bluetooth SDK, but in this example the SPPLEDemo app from the Tiva v1.2 R2 SDK was used running on a DK-TM4C123G.

LightBlue as the Client/SPPLEDemo as the Server

Connecting the Devices

First we need to establish a connection between the devices. To do this open the LightBlue app, you'll seen a screen similar to the following:

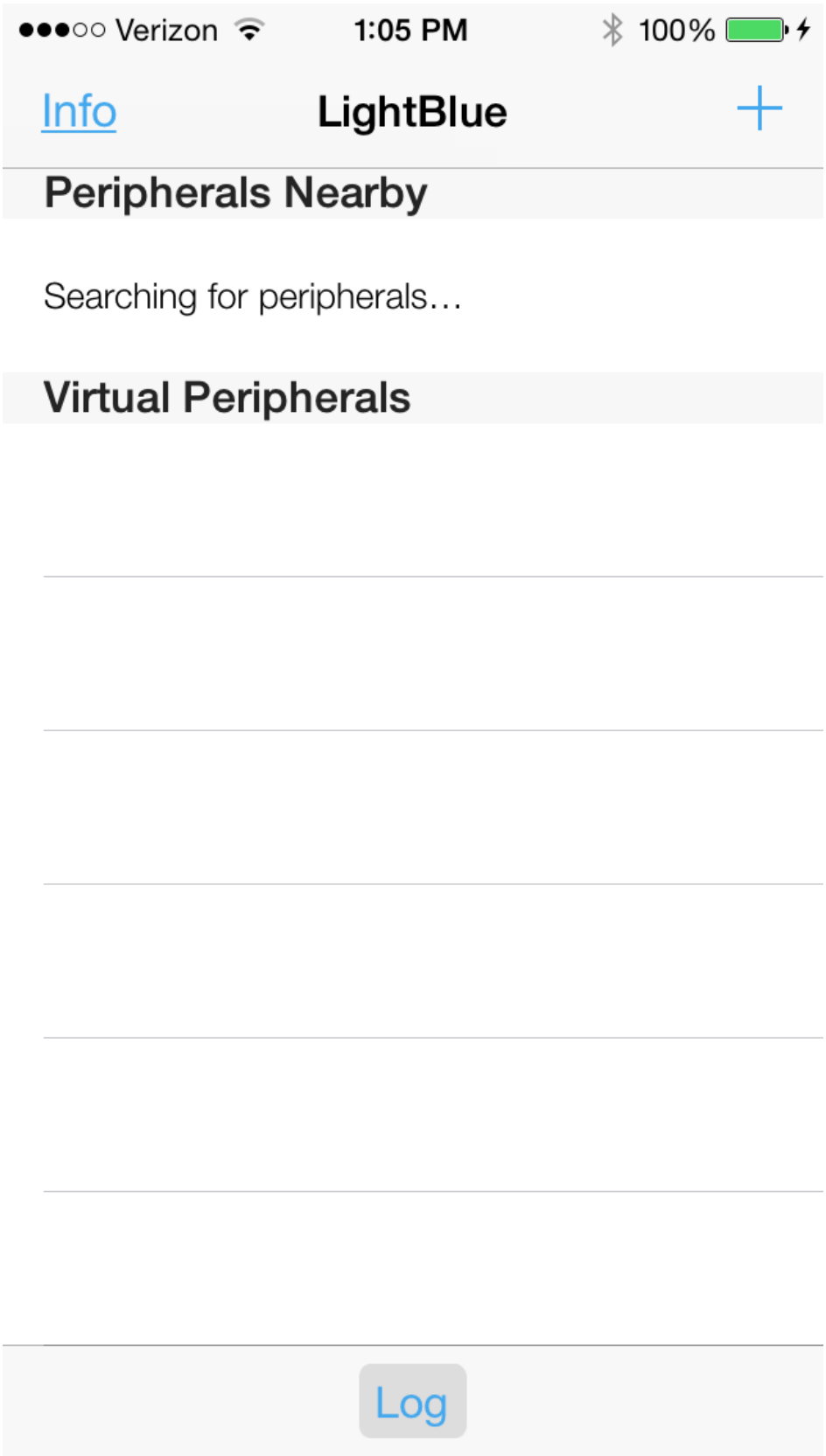

Figure 12-9. SPPLE Demo LightBlue Startup

In the SPPLEDemo terminal start the app as a server, register the SPP LE Service, and begin advertising using the **Server**, **RegisterSPPLE**, and **AdvertiseLE 1** commands. You will see the following in the terminal:

```
OpenStack().
Bluetooth Stack ID: 1.
Device Chipset: 4.1.
BD_ADDR: 0x0017e9d3581a

*****************************************************************
* Command Options: Server, Client, Help                         *
*****************************************************************

SPP+LE>Server

*****************************************************************
* Command Options General: Help, GetLocalAddress, SetBaudRate    *
*                          Quit,                                *
* Command Options BR/EDR:  Inquiry, DisplayInquiryList, Pair,    *
*                          EndPairing, PINCodeResponse,         *
*                          PassKeyResponse,                     *
*                          UserConfirmationResponse,            *
*                          SetDiscoverabilityMode,              *
*                          SetConnectabilityMode,               *
*                          SetPairabilityMode,                  *
*                          ChangeSimplePairingParameters,       *
*                          GetLocalName, SetLocalName,          *
*                          GetClassOfDevice, SetClassOfDevice,  *
*                          GetRemoteName, SniffMode,            *
*                          ExitSniffMode, Open, Close, Read,    *
*                          Write, GetConfigParams,              *
*                          SetConfigParams, GetQueueParams,     *
*                          SetQueueParams, Loopback,            *
*                          DisplayRawModeData, AutomaticReadMode,*
*                          CBSend.                              *
* Command Options GAPLE:   SetDiscoverabilityMode,              *
*                          SetConnectabilityMode,               *
*                          SetPairabilityMode,                  *
*                          ChangePairingParameters,             *
*                          AdvertiseLE, StartScanning,          *
*                          StopScanning, ConnectLE,             *
*                          DisconnectLE, PairLE,                *
*                          LEPasskeyResponse,                   *
*                          QueryEncryptionMode, SetPasskey,     *
*                          DiscoverGAPS, GetLocalName,          *
*                          SetLocalName, GetLERemoteName,       *
*                          SetLocalAppearance,                  *
*                          GetLocalAppearance,                  *
*                          GetRemoteAppearance,                 *
* Command Options SPPLE:   DiscoverSPPLE, RegisterSPPLE, LESend, *
*                          ConfigureSPPLE, LERead, Loopback,    *
*                          DisplayRawModeData, AutomaticReadMode *
*****************************************************************

SPP+LE>RegisterSPPLE
Sucessfully registered SPPLE Service.

SPP+LE>AdvertiseLE 1
   GAP_LE_Advertising_Enable success.
```

Now that SPPLEDemo is advertising you will see the device shown in LightBlue:

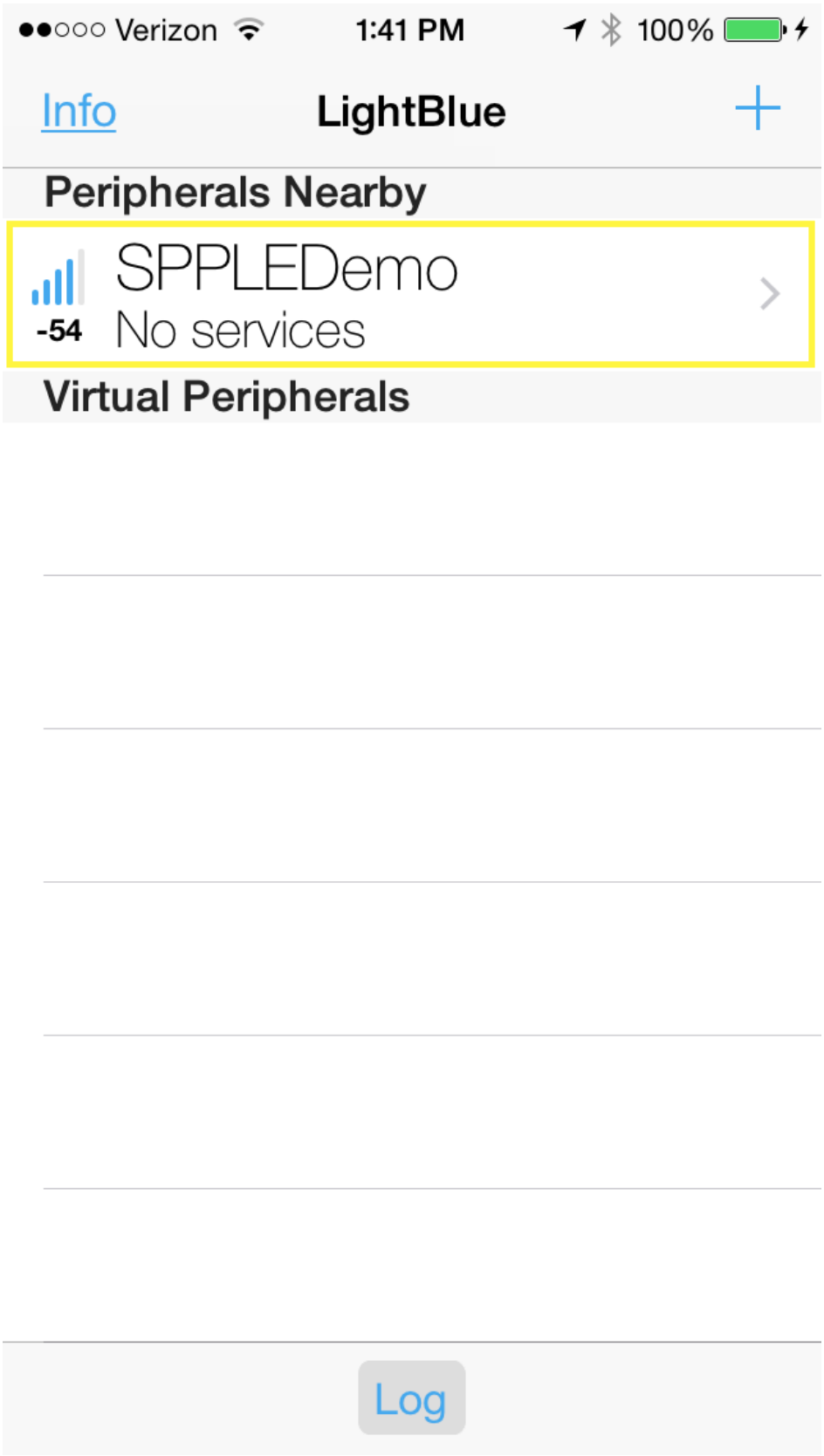

Figure 12-10. SPPLE Demo Discovered by LightBlue

Next select the **SPPLEDemo** device in LightBlue, after doing so you should see the following screen:

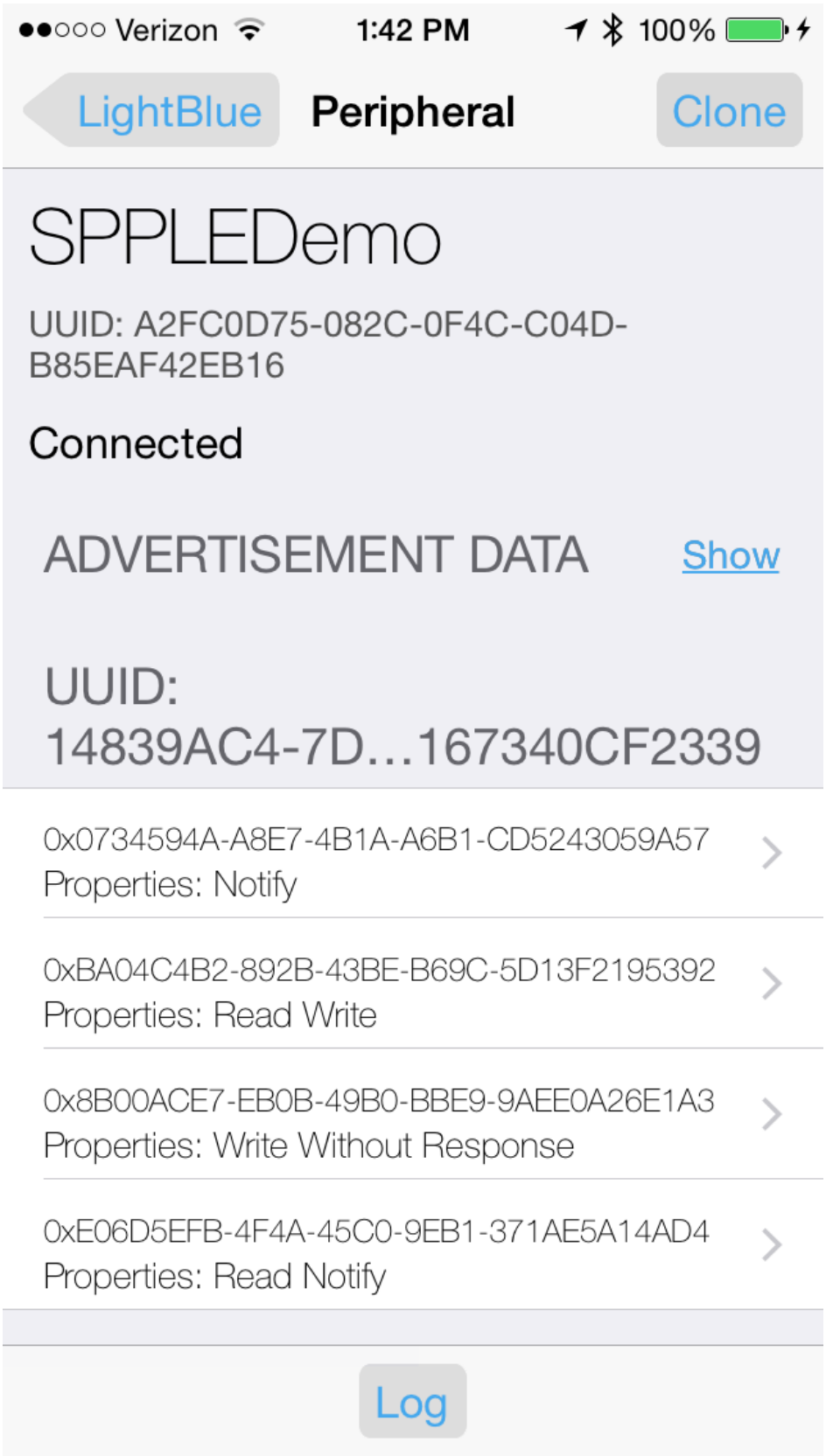

Figure 12-11. SPPLE Demo Connected LightBlue

In the SPPLEDemo terminal you will see the following:

```
etLE_Connection_Complete with size 16.
   Status:       0x00.
   Role:         Slave.
   Address Type: Random.
   BD_ADDR:      0x5cfc3252180b.

SPP+LE>
etGATT_Connection_Device_Connection with size 16:
   Connection ID:   2.
   Connection Type: LE.
   Remote Device:   0x5cfc3252180b.
   Connection MTU:  23.
```

The devices are now connected.

Enabling Notifications

Next enable notifications on the Tx Characteristic and Rx Credits Characteristic in LightBlue by doing the following:

1. Opening the Tx Characteristic **(0x0734594A-A8E7-4B1A-A6B1-CD5243059A57)**.
2. Choosing **Listen for notifications**.
3. Press the back button in the top left corner.
4. Opening the Rx Credits Characteristic **(0xE06D5EFB-4F4A-45C0-9EB1-371AE5A14AD4)**.
5. Choosing **Listen for notifications**.
6. Pressing the back button in the top left corner.

You will notice that after enabling notifications on the Rx Credits Characteristic (**0xE06D5EFB-4F4A-45C0-9EB1-371AE5A14AD4**) that SPPLEDemo sends its initial credits to LightBlue and you will see **0x8300** displayed twice in the app:

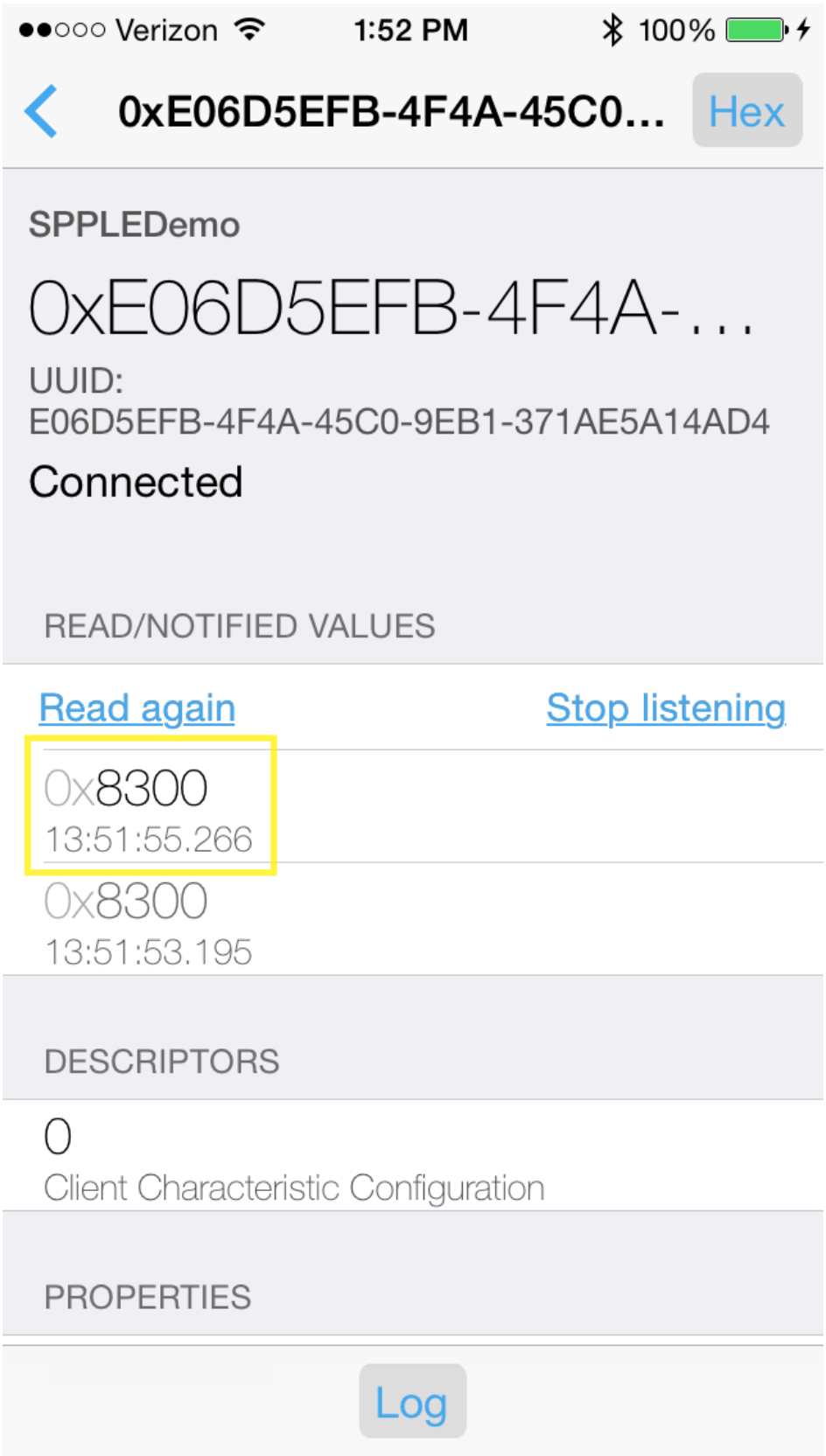

Figure 12-12. SPPLE Demo LightBlue Initial Credits Received

Note

The first instance of 0x8300 is seen because LightBlue read the characteristic automatically when the connection was first established. The data here is displayed in little-endian byte order, the actual number of credits is **0083** in hexadecimal, **131** in decimal.

Sending Data from LightBlue/Receiving Data in SPPLEDemo

At this point the client (LightBlue) can send data to the server (SPPLEDemo). To send data from LightBlue to SPPLEDemo do the following:

1. 1. Open the Rx Characteristic **(0x8B00ACE7-EB0B-49B0-BBE9-9AEE0A26E1A3)**.
2. Choose **Write new value**.
3. Type **414243** (ABC in ASCII).
4. Choose Done.

In the SPPLEDemo terminal you will see a data indication event. To read the data run the **LERead 5cfc3252180b** command, you will see the following in the terminal:

```
Data Indication Event, Connection ID 1, Received 3 bytes.

SPP+LE>LERead 5cfc3252180b
Read: 3.
ABC
```

Now if you open the Rx Credits Characteristic (**0xE06D5EFB-4F4A-45C0-9EB1-371AE5A14AD4**) you will see that SPPLEDemo has credited LightBlue with 3 more credits:

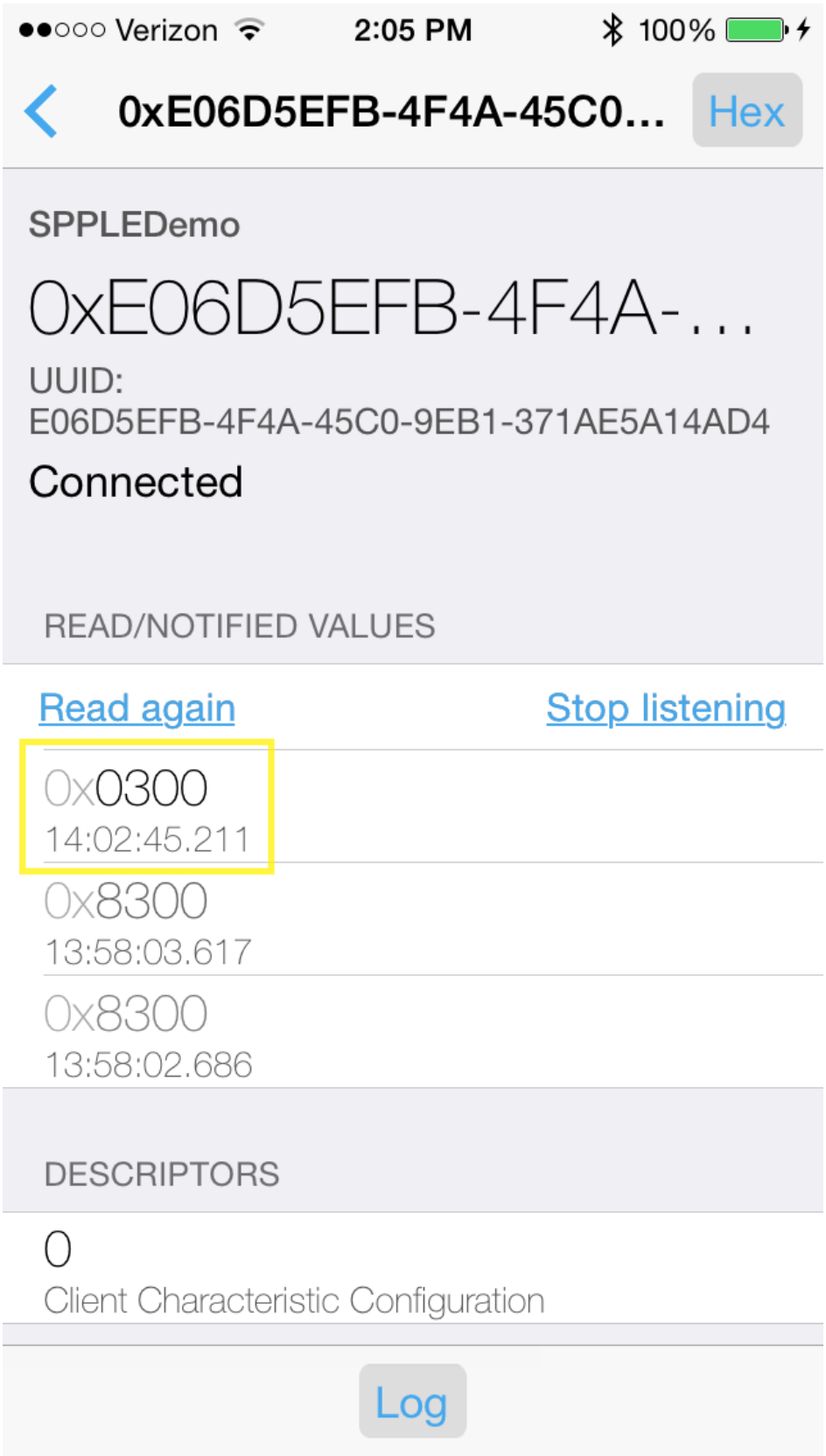

Figure 12-13. SPPL Demo LightBlue Additional Credits Received

Sending Data from SPPLEDemo/Receiving Data in LightBlue

Now we will send data from SPPLEDemo to LightBlue. First LightBlue needs to provide SPPLEDemo with transmit credits. To provide SPPLEDemo with transmit credits do the following in LightBlue:

1. Open the Tx Credits Characteristic **(0xBA04C4B2-892B-43BE-B69C-5D13F2195392)**.
2. Choose "Write new value".
3. Type **6400**. (100 credits = 0x0064 little-endian)
4. Choose **Done**.
5. Press the back button in the top left corner.

We have now given SPPLEDemo 100 credits. Now we can send data in SPPLEDemo using the **LESend 5cfc3252180b 100** command. You should see the following in the terminal:

```
SPP+LE>LESend 5cfc3252180b 100

Send Complete, Sent 100.
```

You can check that LightBlue received the data by:

1. Opening the Tx Characteristic **(0x0734594A-A8E7-4B1A-A6B1-CD5243059A57)**.
2. You will see a long 0x30313233... string of the received data in the list of **NOTIFIED VALUES** as seen below

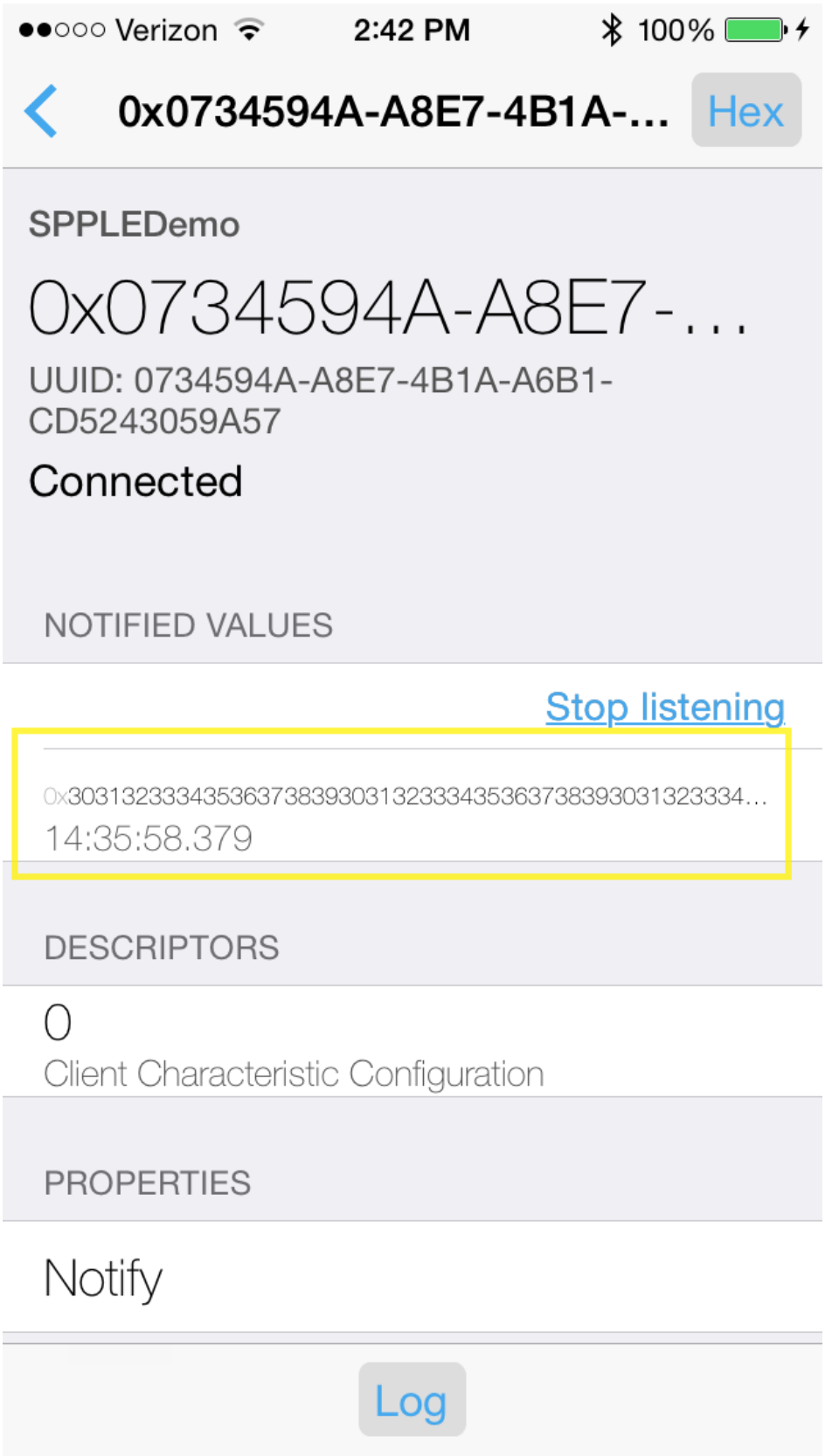

Figure 12-14. SPPLE Demo LightBlue Read Data

Now that LightBlue has received the data it needs to return the transmit credits to SPPLEDemo. This can be done by repeating the sequence above and re-writing **0x6400** to the Tx Credits Characteristic (**0xBA04C4B2-892B-43BE-B69C-5D13F2195392**).

LightBlue as the Server/SPPLEDemo as the Client

Note

LightBlue in the server role does not support displaying the updated value of a characteristic when it is written to. Therefore we will not be able to send data from LightBlue to SPPLEDemo, SPPLEDemo will be able to send data to LightBlue, but that data will not be displayed in the app. This is a limitation of LightBlue.

Connecting the Devices

The first step to connecting the devices is to add the SPP LE Service and its characteristics to LightBlue. It is possible to do this manually by creating a blank virtual peripheral in LightBlue and then adding the necessary service and characteristics, however, it's easier to simply clone SPPLEDemo when it is acting as the server. To clone SPPLEDemo first connect the 2 devices as described above. After the 2 devices are connected choose the **Clone** option in the top right corner of the display. The app will return to the devices list and you will now see SPPLEDemo listed as a Virtual Peripheral as seen below:

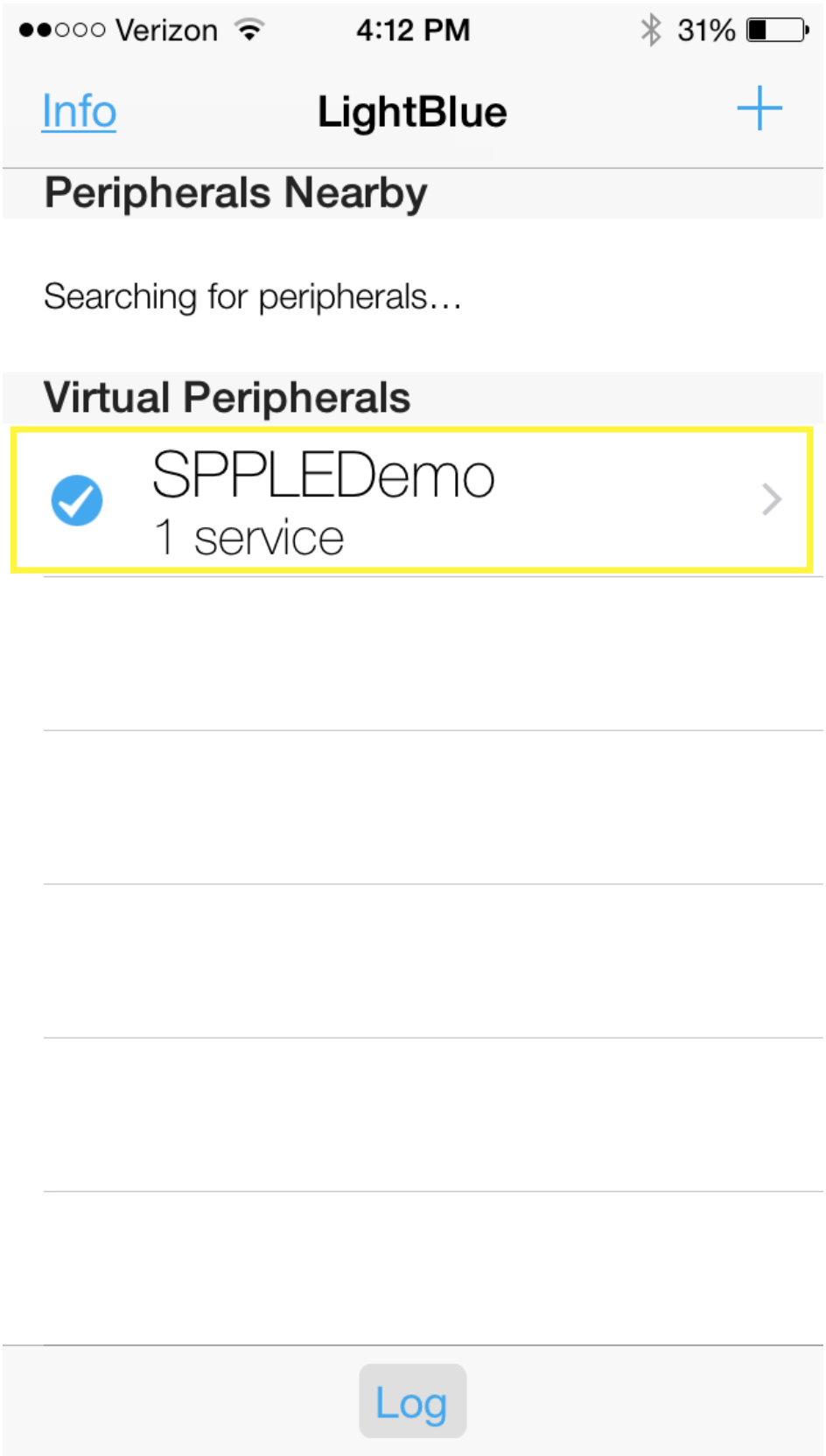

Figure 12-15. SPPLE Demo LightBlue Virtual Peripheral

Note

Make sure that the check box to the left of SPPLEDemo is checked, as seen in the image above. If it is not checked the iDevice is not advertising and SPPLEDemo will not be able to connectd to it.

Now that we have cloned the SPP LE service we can now continue with connecting the devices. Next, restart SPPLEDemo and when prompted start the app as a client. Next scan for the iOS device using the **StartScanning** command. When the iOS device has been found stop the scan using the **StopScanning** command. Now we can connect to the iOS device using the **ConnectLE 5c75524c733a 1** command. After this you have about 10 seconds to run the **DiscoverSPPLE 5c75524c733a** command. After the SPP LE service discovery completes you have about 25 seconds to run the **ConfigureSPPLE 5c75524c733a**. If you wait longer than these times the iOS device will disconnect from SPPLEDemo. After the SPP LE characteristics are configured the 2 apps will stay connected, however, note that if the iOS device goes to sleep it will still close the connection. After running the commands just described you will see output similar to the following in SPPLEDemo's terminal:

```
OpenStack().
Bluetooth Stack ID: 1.
Device Chipset: 4.1.
BD_ADDR: 0xd03972cdab68

*****************************************************************
* Command Options: Server, Client, Help                         *
*****************************************************************

SPP+LE>Client

*****************************************************************
* Command Options General: Help, GetLocalAddress, SetBaudRate    *
*                          Quit,                                *
* Command Options BR/EDR:  Inquiry, DisplayInquiryList, Pair,    *
*                          EndPairing, PINCodeResponse,         *
*                          PassKeyResponse,                     *
*                          UserConfirmationResponse,            *
*                          SetDiscoverabilityMode,              *
*                          SetConnectabilityMode,               *
*                          SetPairabilityMode,                  *
*                          ChangeSimplePairingParameters,       *
*                          GetLocalName, SetLocalName,          *
*                          GetClassOfDevice, SetClassOfDevice,  *
*                          GetRemoteName, SniffMode,            *
*                          ExitSniffMode, Open, Close, Read,    *
*                          Write, GetConfigParams,              *
*                          SetConfigParams, GetQueueParams,     *
*                          DisplayRawModeData, AutomaticReadMode,*
*                          SetQueueParams, Loopback,            *
*                          CBSend.                              *
* Command Options GAPLE:   SetDiscoverabilityMode,              *
*                          SetConnectabilityMode,               *
*                          SetPairabilityMode,                  *
*                          ChangePairingParameters,             *
*                          AdvertiseLE, StartScanning,          *
*                          StopScanning, ConnectLE,             *
*                          DisconnectLE, PairLE,                *
*                          LEPasskeyResponse,                   *
*                          QueryEncryptionMode, SetPasskey,     *
*                          DiscoverGAPS, GetLocalName,          *
*                          SetLocalName, GetLERemoteName,       *
*                          SetLocalAppearance,                  *
*                          GetLocalAppearance,                  *
*                          GetRemoteAppearance,                 *
* Command Options SPPLE:   DiscoverSPPLE, RegisterSPPLE, LESend,*
*                          ConfigureSPPLE, LERead, Loopback,    *
*                          DisplayRawModeData, AutomaticReadMode *
*****************************************************************

SPP+LE>StartScanning
Scan started successfully.

SPP+LE>
etLE_Advertising_Report with size 36.
  1 Responses.
  Advertising Type: rtConnectableUndirected.
```

```
  Address Type: atRandom.
  Address: 0x5c75524c733a.
  RSSI: -71.
  Data Length: 21.
  AD Type: 0x01.
  AD Length: 0x01.
  AD Data: 0x1a
  AD Type: 0x07.
  AD Length: 0x10.
  AD Data: 0x39 0x23 0xcf 0x40 0x73 0x16 0x42 0x9a 0x5c 0x41 0x7e 0x7d 0xc4 0x9a 0x83 0x14

SPP+LE>
etLE_Advertising_Report with size 36.
  1 Responses.
  Advertising Type: rtScanResponse.
  Address Type: atRandom.
  Address: 0x5c75524c733a.
  RSSI: -71.
  Data Length: 11.
  AD Type: 0x09.
  AD Length: 0x09.
  AD Data: 0x53 0x50 0x50 0x4c 0x45 0x44 0x65 0x6d 0x6f

SPP+LE>StopScanning
Scan stopped successfully.

SPP+LE>ConnectLE 5c75524c733a 1
Connection Request successful.

SPP+LE>
etLE_Connection_Complete with size 16.
   Status:       0x00.
   Role:         Master.
   Address Type: Random.
   BD_ADDR:      0x5c75524c733a.

SPP+LE>
etGATT_Connection_Device_Connection with size 16:
   Connection ID:   1.
   Connection Type: LE.
   Remote Device:   0x5c75524c733a.
   Connection MTU:  23.

SPP+LE>
Exchange MTU Response.
Connection ID:   1.
Transaction ID:  1.
Connection Type: LE.
BD_ADDR:         0x5c75524c733a.
MTU:             131.

SPP+LE>
SPP+LE>DiscoverSPPLE 5c75524c733a
GATT_Start_Service_Discovery success.

SPP+LE>
Service 0x000f - 0x001b, UUID: 14839ac47d7e415c9a42167340cf2339.


SPP+LE>
Service Discovery Operation Complete, Status 0x00.

SPP+LE>ConfigureSPPLE 5c75524c733a
SPPLE Service found on remote device, attempting to read Transmit Credits, and configured CCCDs.

SPP+LE>
Write Response.
Connection ID:   1.
Transaction ID:  15.
Connection Type: LE.
BD_ADDR:         0x5c75524c733a.
Bytes Written:   2.

SPP+LE>
Write Response.
Connection ID:   1.
Transaction ID:  16.
Connection Type: LE.
```

```
BD_ADDR:          0x5c75524c733a.
Bytes Written:    2.
```

Note

When SPPLEDemo was acting as the server we had to manually enable notifications with the LightBlue app, however, SPPLEDemo handles enabling notifications automatically when the **ConfigureSPPLE** command is run and this has already been taken care of.

Now that the 2 devices are connected and configured we can now send and receive data between them. Now select the SPPLEDemo Virtual Peripheral in LightBlue to see the virtual peripheral's characteristics. You will see the following or similar on your iDevice's display:

Figure 12-16. SPPLE Demo LightBlue Characteristics

Sending Data from LightBlue/Receiving Data in SPPLEDemo

At this point SPPLEDemo has provided LightBlue with transmit credits and did so when the ConfigureSPPLE command was ran. You should be able to confirm this by opening the SPPLEDemo Virtual Peripheral and choosing the Credits Characteristic (**0xBA04C4B2-892B-43BE-B69C-5D13F2195392**), however, as mentioned above LightBlue does not show updated values of characteristics when they are written to and we have no way to confirm that LightBlue received the data. Even though we can't confirm that LightBlue has received transmit credits, we can still send data from LightBlue to SPPLEDemo. This is true because LightBlue is primarily only a GATT Profile demonstration, it doesn't have any knowledge of the SPP LE protocol that we are using. It is unaware of the transmit credits it has or doesn't have, and, for this reason, we can send data from LightBlue to SPPLEDemo with or without transmit credits. To send data to SPPLEDemo use the Tx Characteristic (**0x0734594A-A8E7-4B1A-A6B1-CD5243059A57'** *) and do the following in LightBlue:*

1. Open the Tx Characteristic and choose the **No value/hex** option.
2. Type in **414243**.
3. Choose **Done**.

In SPPLEDemo you will see a data indication. To read the data use the **LERead 5c75524c733a** command. You should see **ABC** displayed in the terminal, as seen below:

```
Data Indication Event, Connection ID 1, Received 3 bytes.

SPP+LE>LERead 5c75524c733a
Read: 3.
ABC
```

Sending Data from SPPLEDemo/Receiving Data in LightBlue

Note

As mentioned earlier LightBlue does not support showing the updated value of a characteristic when it is written to. We can send data to LightBlue, however, we have no way confirm the data was received.

To send data from SPPLEDemo, LightBlue must first provide it with credits. This can be done using the following in LightBlue:

1. Open the Rx Credits Characteristic **(0xE06D5EFB-4F4A-45C0-9EB1-371AE5A14AD4)**.
2. Type in **6400**. (100 credits = 0x0064 little-endian)
3. Choose Done.

SPPLEDemo now has 100 transmit credits. Next, to send data in SPPLEDemo use the **LESend 5c75524c733a 100** command. You will see the following in your terminal.

```
SPP+LE>LESend 5c75524c733a 100

Send Complete, Sent 100.
```

As mentioned earlier we have no way confirm that LightBlue actually received the data, this is a LightBlue limitation.

LightBlue as the Server/SPPLEDemo as the Client

Note

LightBlue in the server role does not support displaying the updated value of a characteristic when it is written to. Therefore we will not be able to send data from LightBlue to SPPLEDemo, SPPLEDemo will be able to send data to LightBlue, but that data will not be displayed in the app. This is a limitation of LightBlue.

Connecting the Devices

The first step to connecting the devices is to add the SPP LE Service and its characteristics to LightBlue. It is possible to do this manually by creating a blank virtual peripheral in LightBlue and then adding the necessary service and characteristics, however, it's easier to simply clone SPPLEDemo when it is acting as the server. To clone SPPLEDemo first connect the 2 devices as described above. After the 2 devices are connected choose the **Clone** option in the top right corner of the display. The app will return to the devices list and you will now see SPPLEDemo listed as a Virtual Peripheral as seen below:

Figure 12-17. SPPLE Demo LightBlue Virtual Peripheral

Note

Make sure that the check box to the left of SPPLEDemo is checked, as seen in the image above. If it is not checked the iDevice is not advertising and SPPLEDemo will not be able to connectd to it.

Now that we have cloned the SPP LE service we can now continue with connecting the devices. Next, restart SPPLEDemo and when prompted start the app as a client. Next scan for the iOS device using the **StartScanning** command. When the iOS device has been found stop the scan using the **StopScanning** command. Now we can connect to the iOS device using the **ConnectLE 5c75524c733a 1** command. After this you have about 10 seconds to run the **DiscoverSPPLE 5c75524c733a** command. After the SPP LE service discovery completes you have about 25 seconds to run the **ConfigureSPPLE 5c75524c733a**. If you wait longer than these times the iOS device will disconnect from SPPLEDemo. After the SPP LE characteristics are configured the 2 apps will stay connected, however, note that if the iOS device goes to sleep it will still close the connection. After running the commands just described you will see output similar to the following in SPPLEDemo's terminal:

```
OpenStack().
Bluetooth Stack ID: 1.
Device Chipset: 4.1.
BD_ADDR: 0xd03972cdab68

*****************************************************************
* Command Options: Server, Client, Help                         *
*****************************************************************

SPP+LE>Client

*****************************************************************
* Command Options General: Help, GetLocalAddress, SetBaudRate    *
*                          Quit,                                *
* Command Options BR/EDR:  Inquiry, DisplayInquiryList, Pair,    *
*                          EndPairing, PINCodeResponse,         *
*                          PassKeyResponse,                     *
*                          UserConfirmationResponse,            *
*                          SetDiscoverabilityMode,              *
*                          SetConnectabilityMode,               *
*                          SetPairabilityMode,                  *
*                          ChangeSimplePairingParameters,       *
*                          GetLocalName, SetLocalName,          *
*                          GetClassOfDevice, SetClassOfDevice,  *
*                          GetRemoteName, SniffMode,            *
*                          ExitSniffMode, Open, Close, Read,    *
*                          Write, GetConfigParams,              *
*                          SetConfigParams, GetQueueParams,     *
*                          DisplayRawModeData, AutomaticReadMode,*
*                          SetQueueParams, Loopback,            *
*                          CBSend.                              *
* Command Options GAPLE:   SetDiscoverabilityMode,              *
```

```
*                           SetConnectabilityMode,                *
*                           SetPairabilityMode,                   *
*                           ChangePairingParameters,              *
*                           AdvertiseLE, StartScanning,           *
*                           StopScanning, ConnectLE,              *
*                           DisconnectLE, PairLE,                 *
*                           LEPasskeyResponse,                    *
*                           QueryEncryptionMode, SetPasskey,      *
*                           DiscoverGAPS, GetLocalName,           *
*                           SetLocalName, GetLERemoteName,        *
*                           SetLocalAppearance,                   *
*                           GetLocalAppearance,                   *
*                           GetRemoteAppearance,                  *
* Command Options SPPLE:    DiscoverSPPLE, RegisterSPPLE, LESend, *
*                           ConfigureSPPLE, LERead, Loopback,     *
*                           DisplayRawModeData, AutomaticReadMode *
************************************************************************

SPP+LE>StartScanning
Scan started successfully.

SPP+LE>
etLE_Advertising_Report with size 36.
  1 Responses.
  Advertising Type: rtConnectableUndirected.
  Address Type: atRandom.
  Address: 0x5c75524c733a.
  RSSI: -71.
  Data Length: 21.
  AD Type: 0x01.
  AD Length: 0x01.
  AD Data: 0x1a
  AD Type: 0x07.
  AD Length: 0x10.
  AD Data: 0x39 0x23 0xcf 0x40 0x73 0x16 0x42 0x9a 0x5c 0x41 0x7e 0x7d 0xc4 0x9a 0x83 0x14

SPP+LE>
etLE_Advertising_Report with size 36.
  1 Responses.
  Advertising Type: rtScanResponse.
  Address Type: atRandom.
  Address: 0x5c75524c733a.
  RSSI: -71.
  Data Length: 11.
  AD Type: 0x09.
  AD Length: 0x09.
  AD Data: 0x53 0x50 0x50 0x4c 0x45 0x44 0x65 0x6d 0x6f

SPP+LE>StopScanning
Scan stopped successfully.

SPP+LE>ConnectLE 5c75524c733a 1
Connection Request successful.

SPP+LE>
etLE_Connection_Complete with size 16.
   Status:       0x00.
   Role:         Master.
   Address Type: Random.
   BD_ADDR:      0x5c75524c733a.

SPP+LE>
etGATT_Connection_Device_Connection with size 16:
   Connection ID:   1.
   Connection Type: LE.
   Remote Device:   0x5c75524c733a.
   Connection MTU:  23.

SPP+LE>
Exchange MTU Response.
Connection ID:   1.
Transaction ID:  1.
Connection Type: LE.
BD_ADDR:         0x5c75524c733a.
MTU:             131.

SPP+LE>
SPP+LE>DiscoverSPPLE 5c75524c733a
GATT_Start_Service_Discovery success.
```

```
SPP+LE>
Service 0x000f - 0x001b, UUID: 14839ac47d7e415c9a42167340cf2339.


SPP+LE>
Service Discovery Operation Complete, Status 0x00.

SPP+LE>ConfigureSPPLE 5c75524c733a
SPPLE Service found on remote device, attempting to read Transmit Credits, and configured CCCDs.

SPP+LE>
Write Response.
Connection ID:   1.
Transaction ID:  15.
Connection Type: LE.
BD_ADDR:         0x5c75524c733a.
Bytes Written:   2.

SPP+LE>
Write Response.
Connection ID:   1.
Transaction ID:  16.
Connection Type: LE.
BD_ADDR:         0x5c75524c733a.
Bytes Written:   2.
```

Note

When SPPLEDemo was acting as the server we had to manually enable notifications with the LightBlue app, however, SPPLEDemo handles enabling notifications automatically when the **ConfigureSPPLE** command is run and this has already been taken care of.

Now that the 2 devices are connected and configured we can now send and receive data between them. Now select the SPPLEDemo Virtual Peripheral in LightBlue to see the virtual peripheral's characteristics. You will see the following or similar on your iDevice's display:

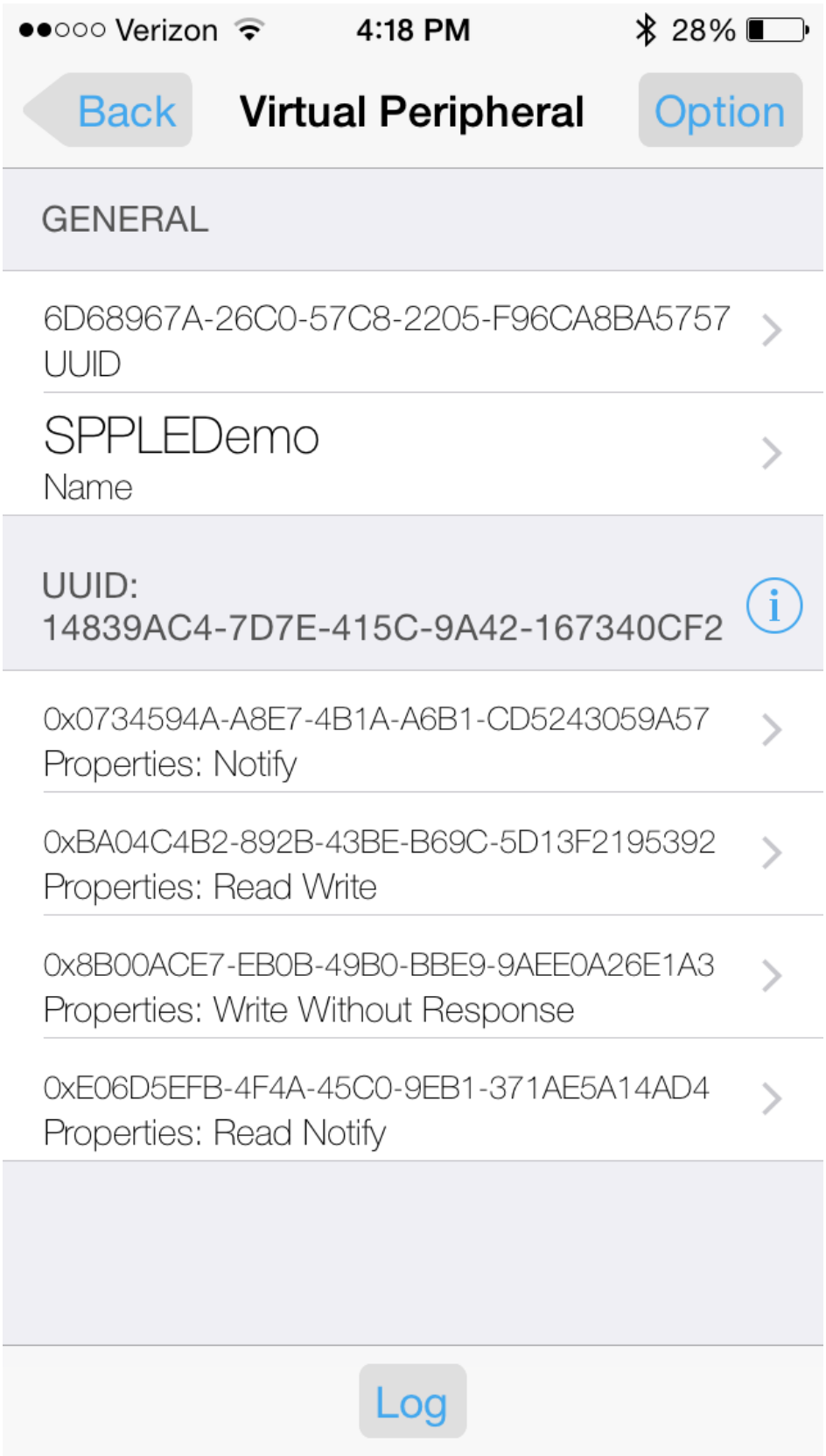

Figure 12-18. SPPLE Demo LightBlue Characteristics

Sending Data from LightBlue/Receiving Data in SPPLEDemo

At this point SPPLEDemo has provided LightBlue with transmit credits and did so when the ConfigureSPPLE command was ran. You should be able to confirm this by opening the SPPLEDemo Virtual Peripheral and choosing the Credits Characteristic (**0xBA04C4B2-892B-43BE-B69C-5D13F2195392**), however, as mentioned above LightBlue does not show updated values of characteristics when they are written to and we have no way to confirm that LightBlue received the data. Even though we can't confirm that LightBlue has received transmit credits, we can still send data from LightBlue to SPPLEDemo. This is true because LightBlue is primarily only a GATT Profile demonstration, it doesn't have any knowledge of the SPP LE protocol that we are using. It is unaware of the transmit credits it has or doesn't have, and, for this reason, we can send data from LightBlue to SPPLEDemo with or without transmit credits. To send data to SPPLEDemo use the Tx Characteristic (**0x0734594A-A8E7-4B1A-A6B1-CD5243059A57'** *) and do the following in LightBlue:*

1. Open the Tx Characteristic and choose the **No value/hex** option.
2. Type in **414243**.
3. Choose **Done**.

In SPPLEDemo you will see a data indication. To read the data use the **LERead 5c75524c733a** command. You should see **ABC** displayed in the terminal, as seen below:

```
Data Indication Event, Connection ID 1, Received 3 bytes.

SPP+LE>LERead 5c75524c733a
Read: 3.
ABC
```

Sending Data from SPPLEDemo/Receiving Data in LightBlue

Note

As mentioned earlier LightBlue does not support showing the updated value of a characteristic when it is written to. We can send data to LightBlue, however, we have no way confirm the data was received.

To send data from SPPLEDemo, LightBlue must first provide it with credits. This can be done using the following in LightBlue:

1. 1. Open the Rx Credits Characteristic **(0xE06D5EFB-4F4A-45C0-9EB1-371AE5A14AD4)**.
2. Type in **6400**. (100 credits = 0x0064 little-endian)
3. Choose **Done**.

SPPLEDemo now has 100 transmit credits. Next, to send data in SPPLEDemo use the **LESend 5c75524c733a 100** command. You will see the following in your terminal.

```
SPP+LE>LESend 5c75524c733a 100

Send Complete, Sent 100.
```

As mentioned earlier we have no way confirm that LightBlue actually received the data, this is a LightBlue limitation.

LightBlue as the Server/SPPLEDemo as the Client

Note

LightBlue in the server role does not support displaying the updated value of a characteristic when it is written to. Therefore we will not be able to send data from LightBlue to SPPLEDemo, SPPLEDemo will be able to send data to LightBlue, but that data will not be displayed in the app. This is a limitation of LightBlue.

Connecting the Devices

The first step to connecting the devices is to add the SPP LE Service and its characteristics to LightBlue. It is possible to do this manually by creating a blank virtual peripheral in LightBlue and then adding the necessary service and characteristics, however, it's easier to simply clone SPPLEDemo when it is acting as the server. To clone SPPLEDemo first connect the 2 devices as described above. After the 2 devices are connected choose

the **Clone** option in the top right corner of the display. The app will return to the devices list and you will now see SPPLEDemo listed as a Virtual Peripheral as seen below:

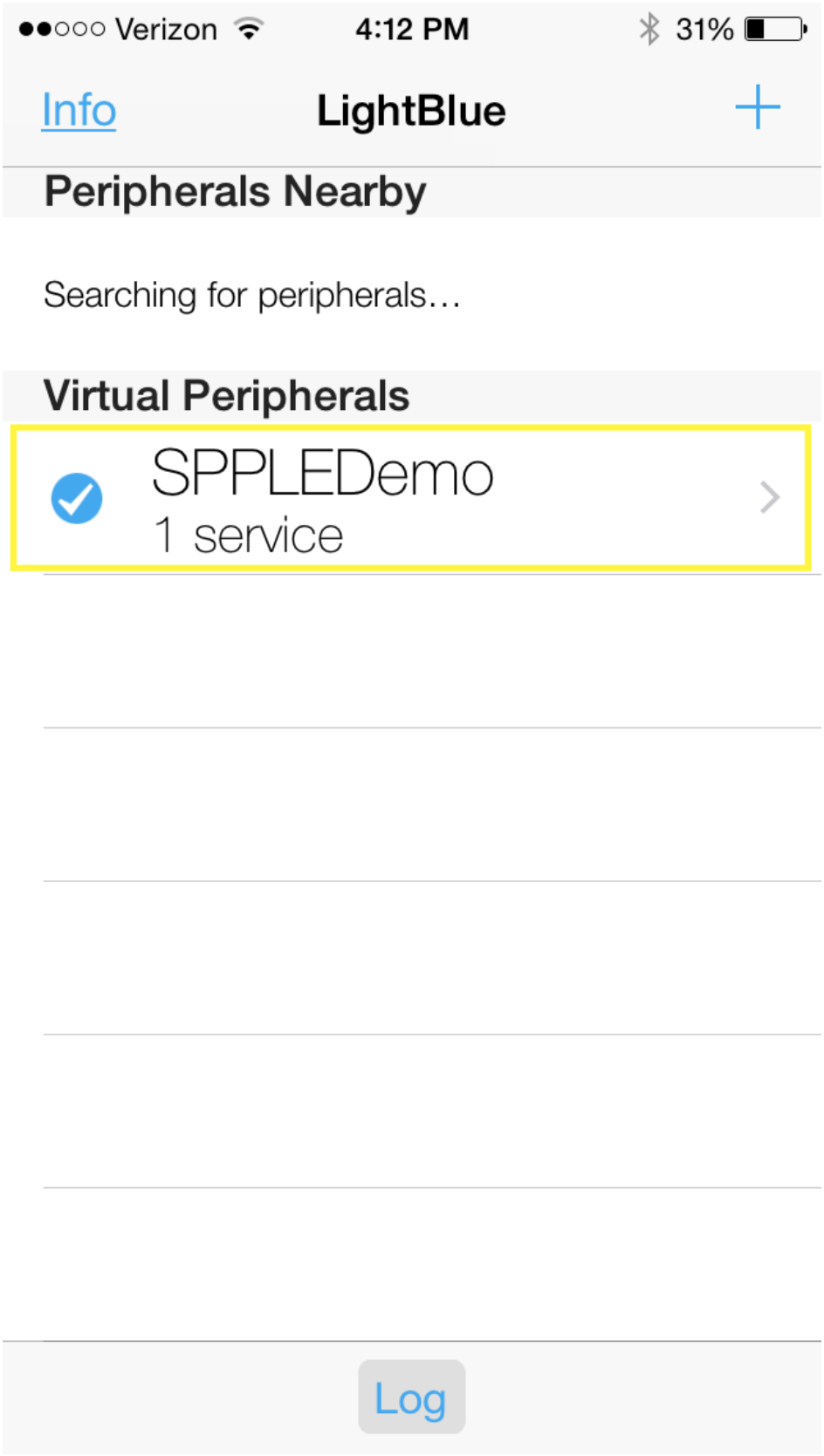

Figure 12-19. SPPLE Demo LightBlue Virtual Peripheral

Note

Make sure that the check box to the left of SPPLEDemo is checked, as seen in the image above. If it is not checked the iDevice is not advertising and SPPLEDemo will not be able to connectd to it.

Now that we have cloned the SPP LE service we can now continue with connecting the devices. Next, restart SPPLEDemo and when prompted start the app as a client. Next scan for the iOS device using the **StartScanning** command. When the iOS device has been found stop the scan using the **StopScanning** command. Now we can connect to the iOS device using the **ConnectLE 5c75524c733a 1** command. After this you have about 10 seconds to run the **DiscoverSPPLE 5c75524c733a** command. After the SPP LE service discovery completes you have about 25 seconds to run the **ConfigureSPPLE 5c75524c733a**. If you wait longer than these times the iOS device will disconnect from SPPLEDemo. After the SPP LE characteristics are configured the 2 apps will stay connected, however, note that if the iOS device goes to sleep it will still close the connection. After running the commands just described you will see output similar to the following in SPPLEDemo's terminal:

```
OpenStack().
Bluetooth Stack ID: 1.
Device Chipset: 4.1.
BD_ADDR: 0xd03972cdab68

*****************************************************************
* Command Options: Server, Client, Help                         *
*****************************************************************

SPP+LE>Client

*****************************************************************
* Command Options General: Help, GetLocalAddress, SetBaudRate    *
*                          Quit,                                *
* Command Options BR/EDR:  Inquiry, DisplayInquiryList, Pair,    *
*                          EndPairing, PINCodeResponse,         *
*                          PassKeyResponse,                     *
*                          UserConfirmationResponse,            *
*                          SetDiscoverabilityMode,              *
*                          SetConnectabilityMode,               *
*                          SetPairabilityMode,                  *
*                          ChangeSimplePairingParameters,       *
*                          GetLocalName, SetLocalName,          *
*                          GetClassOfDevice, SetClassOfDevice,  *
*                          GetRemoteName, SniffMode,            *
*                          ExitSniffMode, Open, Close, Read,    *
*                          Write, GetConfigParams,              *
*                          SetConfigParams, GetQueueParams,     *
*                          DisplayRawModeData, AutomaticReadMode,*
*                          SetQueueParams, Loopback,            *
*                          CBSend.                              *
* Command Options GAPLE:   SetDiscoverabilityMode,              *
*                          SetConnectabilityMode,               *
*                          SetPairabilityMode,                  *
*                          ChangePairingParameters,             *
*                          AdvertiseLE, StartScanning,          *
*                          StopScanning, ConnectLE,             *
*                          DisconnectLE, PairLE,                *
*                          LEPasskeyResponse,                   *
*                          QueryEncryptionMode, SetPasskey,     *
*                          DiscoverGAPS, GetLocalName,          *
*                          SetLocalName, GetLERemoteName,       *
*                          SetLocalAppearance,                  *
*                          GetLocalAppearance,                  *
*                          GetRemoteAppearance,                 *
* Command Options SPPLE:   DiscoverSPPLE, RegisterSPPLE, LESend, *
*                          ConfigureSPPLE, LERead, Loopback,    *
*                          DisplayRawModeData, AutomaticReadMode *
*****************************************************************

SPP+LE>StartScanning
Scan started successfully.

SPP+LE>
etLE_Advertising_Report with size 36.
  1 Responses.
  Advertising Type: rtConnectableUndirected.
```

```
  Address Type: atRandom.
  Address: 0x5c75524c733a.
  RSSI: -71.
  Data Length: 21.
  AD Type: 0x01.
  AD Length: 0x01.
  AD Data: 0x1a
  AD Type: 0x07.
  AD Length: 0x10.
  AD Data: 0x39 0x23 0xcf 0x40 0x73 0x16 0x42 0x9a 0x5c 0x41 0x7e 0x7d 0xc4 0x9a 0x83 0x14

SPP+LE>
etLE_Advertising_Report with size 36.
  1 Responses.
  Advertising Type: rtScanResponse.
  Address Type: atRandom.
  Address: 0x5c75524c733a.
  RSSI: -71.
  Data Length: 11.
  AD Type: 0x09.
  AD Length: 0x09.
  AD Data: 0x53 0x50 0x50 0x4c 0x45 0x44 0x65 0x6d 0x6f

SPP+LE>StopScanning
Scan stopped successfully.

SPP+LE>ConnectLE 5c75524c733a 1
Connection Request successful.

SPP+LE>
etLE_Connection_Complete with size 16.
   Status:       0x00.
   Role:         Master.
   Address Type: Random.
   BD_ADDR:      0x5c75524c733a.

SPP+LE>
etGATT_Connection_Device_Connection with size 16:
   Connection ID:   1.
   Connection Type: LE.
   Remote Device:   0x5c75524c733a.
   Connection MTU:  23.

SPP+LE>
Exchange MTU Response.
Connection ID:   1.
Transaction ID:  1.
Connection Type: LE.
BD_ADDR:         0x5c75524c733a.
MTU:             131.

SPP+LE>
SPP+LE>DiscoverSPPLE 5c75524c733a
GATT_Start_Service_Discovery success.

SPP+LE>
Service 0x000f - 0x001b, UUID: 14839ac47d7e415c9a42167340cf2339.


SPP+LE>
Service Discovery Operation Complete, Status 0x00.

SPP+LE>ConfigureSPPLE 5c75524c733a
SPPLE Service found on remote device, attempting to read Transmit Credits, and configured CCCDs.

SPP+LE>
Write Response.
Connection ID:   1.
Transaction ID:  15.
Connection Type: LE.
BD_ADDR:         0x5c75524c733a.
Bytes Written:   2.

SPP+LE>
Write Response.
Connection ID:   1.
Transaction ID:  16.
Connection Type: LE.
```

```
BD_ADDR:          0x5c75524c733a.
Bytes Written:    2.
```

Note

When SPPLEDemo was acting as the server we had to manually enable notifications with the LightBlue app, however, SPPLEDemo handles enabling notifications automatically when the **ConfigureSPPLE** command is run and this has already been taken care of.

Now that the 2 devices are connected and configured we can now send and receive data between them. Now select the SPPLEDemo Virtual Peripheral in LightBlue to see the virtual peripheral's characteristics. You will see the following or similar on your iDevice's display:

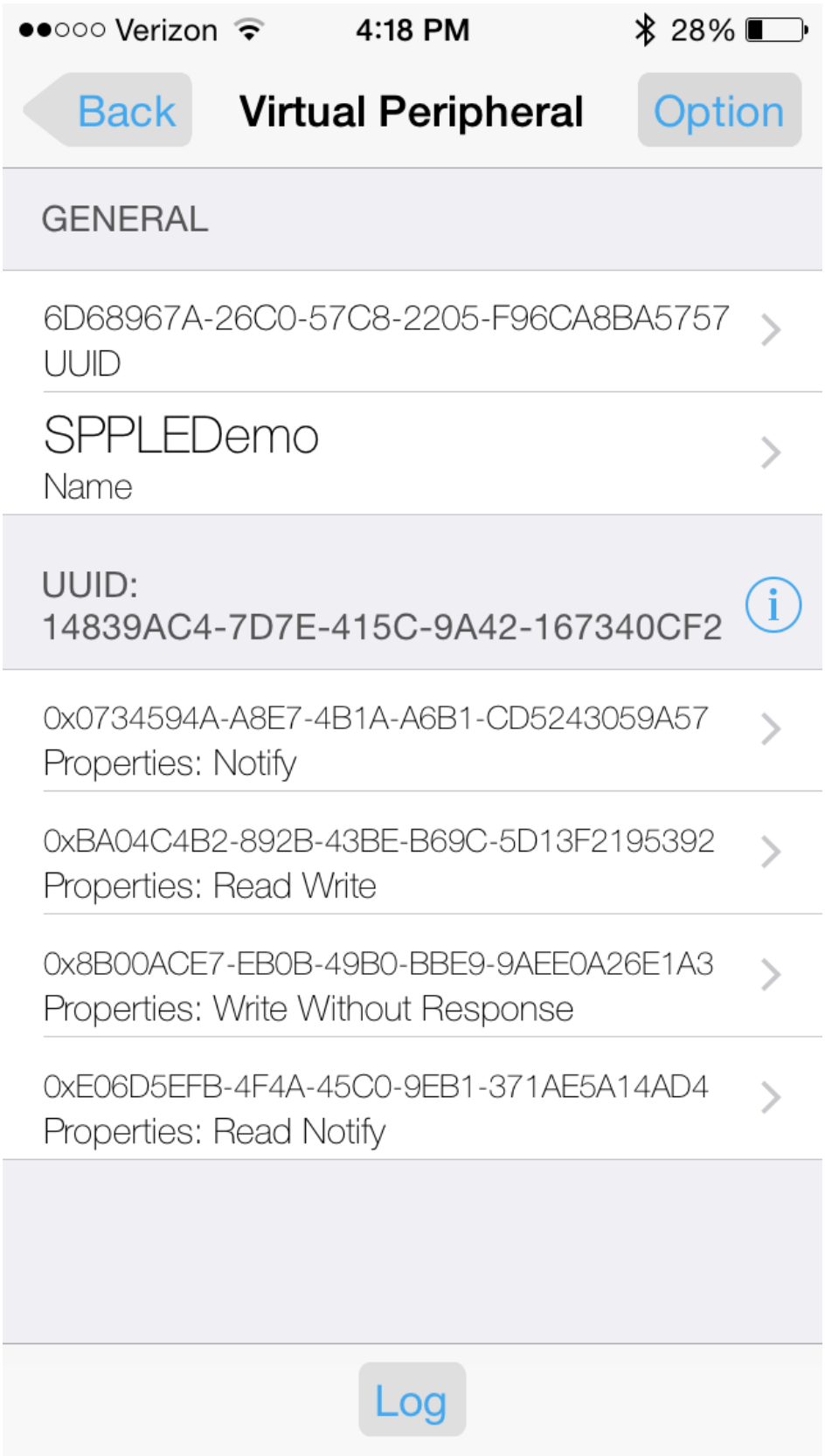

Figure 12-20. SPPLE Demo Characteristics

Sending Data from LightBlue/Receiving Data in SPPLEDemo

At this point SPPLEDemo has provided LightBlue with transmit credits and did so when the ConfigureSPPLE command was ran. You should be able to confirm this by opening the SPPLEDemo Virtual Peripheral and choosing the Credits Characteristic (**0xBA04C4B2-892B-43BE-B69C-5D13F2195392**), however, as mentioned above LightBlue does not show updated values of characteristics when they are written to and we have no way to confirm that LightBlue received the data. Even though we can't confirm that LightBlue has received transmit credits, we can still send data from LightBlue to SPPLEDemo. This is true because LightBlue is primarily only a GATT Profile demonstration, it doesn't have any knowledge of the SPP LE protocol that we are using. It is unaware of the transmit credits it has or doesn't have, and, for this reason, we can send data from LightBlue to SPPLEDemo with or without transmit credits. To send data to SPPLEDemo use the Tx Characteristic (**0x0734594A-A8E7-4B1A-A6B1-CD5243059A57'** *) and do the following in LightBlue:*

1. 1. Open the Tx Characteristic and choose the No value/hex option.
2. Type in 414243.
3. Choose Done.

In SPPLEDemo you will see a data indication. To read the data use the **LERead 5c75524c733a** command. You should see **ABC** displayed in the terminal, as seen below:

```
Data Indication Event, Connection ID 1, Received 3 bytes.

SPP+LE>LERead 5c75524c733a
Read: 3.
ABC
```

Data Indication Event, Connection ID 1, Received 3 bytes. SPP+LE>LERead 5c75524c733a Read: 3. ABC

12.4 Demonstrating SPP LE on an iOS Device with the SPPLE Transfer App - LEGACY

Note

The SPPLE Transfer app is no longer available on the iOS app store as of Q1 2015, however, the LightBlue app is available and can be used as a replacement. Refer to the above section Demonstrating SPP LE on an iOS Device with the LightBlue App for a demonstration of using the LightBlue app with SPPLEDemo.

SPPLE is not a standard Bluetooth Profile. You will have to make sure the app can use the custom UUIDs that are needed to communicate and read and write to the app.

The MSP430 device can also connect to an iPhone running an SPPLE application. The application that we use on the iPhone is SPPLE Transfer (a.k.a. SPPLE Chat) which can be downloaded for free from the app store here. There are some changes that need to be made to the **SPPLEDemo.c** file as well. See below:

- In the function **ConfigureSPPLE** make the following changes to the if statement. After the comment /* **Determine if a service discovery operation has been previously done** */ and before the **else case**.

```
/* Determine if a service discovery operation has been      */
/* previously done.                                         */
   /*changed for using it with SPPLE chat application on iPhone*/
   if(TRUE)
   {
    Display(("SPPLE Service found on remote device, attempting to read Transmit Credits, and
configured CCCDs.\r\n"));

    /* Enable Notifications on the proper characteristics.       */
    EnableDisableNotificationsIndications(LEContextInfo[LEConnectionIndex].ConnectionID, DeviceInfo-
>ClientInfo.Tx_Client_Configuration_Descriptor,
                    GATT_CLIENT_CONFIGURATION_CHARACTERISTIC_NOTIFY_ENABLE,
GATT_ClientEventCallback_SPPLE);

    ret_val = 0;
}
else
```

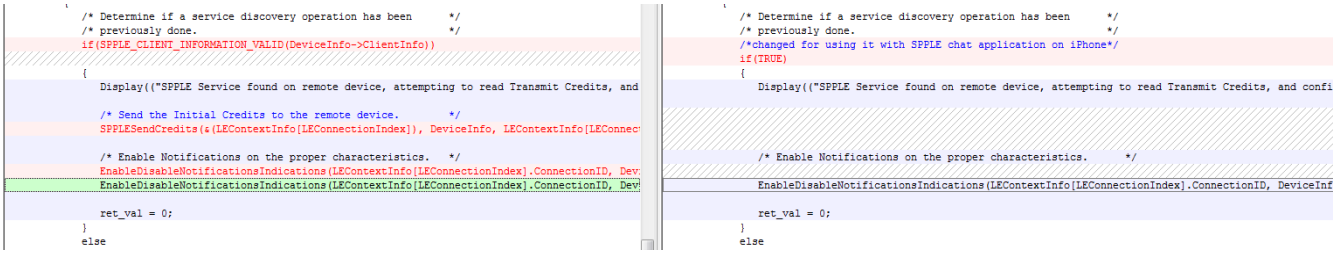

Figure 12-21. File comparison showing changes needed for ConfigureSPPLE

- In the function SendDataCommand add the following code after the SendInfo.BytesSent = 0 and before the comment /* Kick start the send process. */.

```
LEContextInfo[LEConnectionIndex].SPPLEBufferInfo.TransmitCredits = 1000;
DeviceInfo->ServerInfo.Tx_Client_Configuration_Descriptor        =
GATT_CLIENT_CONFIGURATION_CHARACTERISTIC_NOTIFY_ENABLE;
```

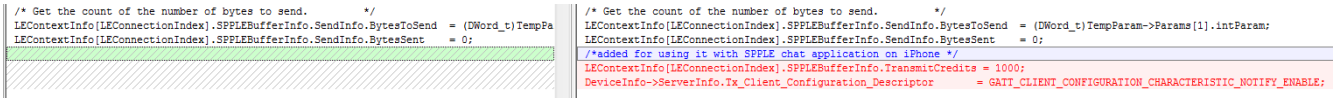

Figure 12-22. File comparision showing changes needed for SendDataCommand

- Load the SPP LE profile on to the MSP430F5438A device by rebuilding the project and flashing it from the project.
- Set up a Terminal Program for the Serial Port that the device is connected to. The serial parameters to use are 115200 Baud, 8, no, 1 and no flow control. Once connected, reset the

```
    device using Reset S3 button and you should see the stack getting initialized on the terminal.
```

- On the iPhone open the SPPLE chat application. Choose peripheral mode and turn on advertising.
- On the MSP430 device, StartScanning to find out devices in the area that are connectable. The Bluetooth address of the Iphone should show up something like this:

```
etLE_Advertising_Report with size 36.
  1 Responses.
  Advertising Type: rtConnectableUndirected.
  Address Type: atRandom.
  Address: 0x79F20C012372.
  RSSI: 0xFFFFFFCB.
  Data Length: 29.
  AD Type: 0x01.
  AD Length: 0x01.
  AD Data: 0x1A
  AD Type: 0x07.
  AD Length: 0x10.
  AD Data: 0x39 0x23 0xCF 0x40 0x73 0x16 0x42 0x9A 0x5C 0x41 0x7E 0x7D 0xC4 0x9A 0x83 0x14
  AD Type: 0x09.
  AD Length: 0x06.
  AD Data: 0x69 0x50 0x68 0x6F 0x6E 0x65

LE>etLE_Advertising_Report with size 36.
  1 Responses.
  Advertising Type: rtScanResponse.
  Address Type: atRandom.
  Address: 0x79F20C012372.
  RSSI: 0xFFFFFFCB.
  Data Length: 0.
```

- The address type will be random. Note down the address of the device specified.
- Connect to the remote device using the Connectle <bd-addr> 1 command where the bd-addr is the previously noted address.
- Discover services using Discoverspple and configure services using Configurespple.

- Now the two devices are connected. Data from the iphone can be send by typing text on the text box and hitting send.
- Data from the MSP device can be sent using the Senddata command. It is read and displayed automatically in the output window of the app.

12.5 SPP Demo

To use classic SPP on this demo, please follow same instructions here: CC256x MSP430 TI's Bluetooth stack Basic SPPDemo APP

12.6 Application Commands

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands

BR/EDR Commands

For BR/EDR Commands refer to the document SPP Profile sections Generic Access Profile Commands and Serial Port Profile Commands.

GAPLE Commands

The Generic Access Profile defines standard procedures related to the discovery and connection of Bluetooth devices. It defines modes of operation that are generic to all devices and allows for procedures which use those modes to decide how a device can be interacted with by other Bluetooth devices. Discoverability, Connectability, Pairability, Bondable Modes, and Security Modes can all be changed using Generic Access Profile procedures. All of these modes affect the interaction two devices may have with one another. GAP also defines the procedures for how bond two Bluetooth devices.

SetDiscoverabilityMode

Description

The SetDiscoverabilityMode command is responsible for setting the Discoverability Mode of the local device. This command returns zero on successful execution and a negative value on all errors. The Discoverability Mode in LE is only applicable when advertising, if a device is not advertising it is not discoverable. The value set by this command will be used as a parameter in the command AdvertiseLE.

Parameters

This command requires only one parameter which is an integer value that represents a Discoverability Mode. This value must be specified as 0 (for Non-Discoverable Mode), 1 (for Limited Discoverable Mode), or 2 (for General Discoverable Mode).

Command Call Examples

"SetDiscoverabilityMode 0" Attempts to change the Discoverability Mode of the Local Device to Non-Discoverable. "SetDiscoverabilityMode 1" Attempts to change the Discoverability Mode of the Local Device to Limited Discoverable. "SetDiscoverabilityMode 2" Attempts to change the Discoverability Mode of the Local Device to General Discoverable.

Possible Return Values

(0) Successfully Set Discoverability Mode Parameter

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

SetConnectabilityMode

Description

The SetConnectabilityMode command is responsible for setting the Connectability Mode of the local device. This command returns zero on successful execution and a negative value on all errors. The Connectability Mode in

LE is only applicable when advertising, if a device is not advertising it is not connectable. The value set by this command will be used as a parameter in the command AdvertiseLE.

Parameters

This command requires only one parameter which is an integer value that represents a Connectability Mode. This value must be specified as 0 (for Non-Connectable) or 1 (for Connectable).

Command Call Examples

"SetConnectabilityMode 0" Attempts to set the Local Device's Connectability Mode to Non-Connectable. "SetConnectabilityMode 1" Attempts to set the Local Device's Connectability Mode to Connectable.

Possible Return Values

(0) Successfully Set Connectability Mode Parameter

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

SetPairabilityMode

Description

The SetPairabilityMode command is responsible for setting the Pairability Mode of the local device. This command returns zero on successful execution and a negative value on all errors.

Parameters

This command requires only one parameter which is an integer value that represents a Pairability Mode. This value must be specified as 0 (for Non-Pairable), 1 (for Pairable) or 2(for Pairable with Secure Simple Pairing).

Command Call Examples

"SetPairabilityMode 0" Attempts to set the Local Device's Pairability Mode to Non-Pairable. "SetPairabilityMode 1" Attempts to set the Local Device's Pairability Mode to Pairable.

Possible Return Values

(0) Successfully Set Pairability Mode

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

API Call

GAP_LE_Set_Pairability_Mode(BluetoothStackID, PairabilityMode);

API Prototype

int BTPSAPI GAP_LE_Set_Pairability_Mode(unsigned int BluetoothStackID, GAP_LE_Pairability_Mode_t PairableMode);

Description of API

This function is provided to allow the local host the ability to change the pairability mode used by the local host. This function will return zero if successful or a negative return error code if there was an error condition.

ChangePairingParameters

Description

The ChangePairingParameters command is responsible for changing the LE Pairing Parameters that are exchanged during the Pairing procedure. This command returns zero on successful execution and a negative value on all errors.

Parameters

This command requires five parameters which are the I/O Capability, the Bonding Type, the MITM Requirement, the SC Enable and the P256 debug mode.

The first parameter must be specified as 0 (for Display Only), 1 (for Display Yes/No), 2 (for Keyboard Only), 3 (for No Input/Output) or 4 (for Keyboard/Display).

The second parameter must be specified as 0 (for No Bonding) or 1 (for Bonding), when at least one of the devices is set to No Bonding, the LTK won't be stored.

The third parameter must be specified as 0 (for No MITM) or 1 (for MITM required).

The fourth parameter must be specified as 0 (for SC disabled) or 1 (for SC enabled), when using SC disable, legacy pairing procedure will take place.

The fifth parameter must be specified as 0 (for Debug Mode disabled) or 1 (for P256 debug mode enabled), Only when using SC pairing, P256 debug mode is relevant and when it is set, the values of the P256 private and public keys will be pre-defined according to the Bluetooth specification instead of random.

Command Call Examples

"ChangeSimplePairingParameters 3 0 0 0 0" Attempts to set the I/O Capability to No Input/Output, Bonding Type set to No Bonding, turns off MITM Protection, Disable secure connections and disable debug mode.

"ChangeSimplePairingParameters 2 0 1 1 0 " Attempts to set the I/O Capability to Keyboard Only, Bonding Type set to No Bonding, activates MITM Protection, Enabling secure connections and disable debug mode.

"ChangeSimplePairingParameters 1 1 1 1 1" Attempts to set the I/O Capability to Display Yes/No, Bonding Type set to Bonding, activates MITM Protection, Enabling secure connections and enabling debug mode.

Possible Return Values

(0) Successfully Set Pairability Mode

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

AdvertiseLE

Description

The AdvertiseLE command is responsible for enabling LE Advertisements. This command returns zero on successful execution and a negative value on all errors.

Parameters

The only parameter necessary decides whether Advertising Reports are sent or are disabled. To Disable, use 0 as the first parameter, to enable, use 1 instead.

Command Call Examples

“AdvertiseLE 1” Attempts to enable Low Energy Advertising on the local Bluetooth device. “AdvertiseLE 0” Attempts to disable Low Energy Advertising on the local Bluetooth device.

Possible Return Values

(0) Successfully Set Pairability Mode

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1) BTPS_ERROR_INVALID_PARAMETER

(-56) BTPS_ERROR_GAP_NOT_INITIALIZED

(-104) BTPS_ERROR_LOCAL_CONTROLLER_DOES_NOT_SUPPORT_LE

(-57) BTPS_ERROR_DEVICE_HCI_ERROR

API Calls

Depending on the First Parameter Value

GAP_LE_Advertising_Disable(BluetoothStackID);

GAP_LE_Set_Advertising_Data(BluetoothStackID,
(Advertisement_Data_Buffer.AdvertisingData.Advertising_Data[0] + 1),
&(Advertisement_Data_Buffer.AdvertisingData));

GAP_LE_Set_Scan_Response_Data(BluetoothStackID,
(Advertisement_Data_Buffer.ScanResponseData.Scan_Response_Data[0] + 1),
&(Advertisement_Data_Buffer.ScanResponseData));

GAP_LE_Advertising_Enable(BluetoothStackID, TRUE, &AdvertisingParameters, &ConnectabilityParameters,
GAP_LE_Event_Callback, 0);

API Prototypes

int BTPSAPI GAP_LE_Advertising_Disable(unsigned int BluetoothStackID);

int BTPSAPI GAP_LE_Set_Advertising_Data(unsigned int BluetoothStackID, unsigned int Length,
*Advertising_Data_t *Advertising_Data);*

int BTPSAPI GAP_LE_Set_Scan_Response_Data(unsigned int BluetoothStackID, unsigned int Length,
*Scan_Response_Data_t *Scan_Response_Data);*

int BTPSAPI GAP_LE_Set_Advertising_Data(unsigned int BluetoothStackID, unsigned int Length,
*Advertising_Data_t *Advertising_Data);*

int BTPSAPI GAP_LE_Set_Advertising_Data(unsigned int BluetoothStackID, unsigned int Length,
*Advertising_Data_t *Advertising_Data);*

Description of API

The GAP_LE_Advertising_Disable function is provided to allow the local host the ability to cancel (stop) an on-going advertising procedure. This function will return zero if successful or a negative return error code if there was an error condition. The GAP_LE_Set_Advertising_Data is provided to allow the local host the ability to set the advertising data that is used during the advertising procedure (started via the GAP_LE_Advertising_Enable function). This function will return zero if successful or a negative return error code if there was an error condition. The GAP_LE_Set_Scan_Response_Data function is provided to allow the local host the ability to set the advertising data that is used during the advertising procedure (started via the GAP_LE_Advertising_Enable function). This function will return zero if successful or a negative return error code if there was an error condition. The GAP_LE_Set_Advertising_Data function is provided to allow the local host the ability to set the advertising data that is used during the advertising procedure (started via the GAP_LE_Advertising_Enable function). This function will return zero if successful or a negative return error code if there was an error condition.

StartScanning

Description

The StartScanning command is responsible for starting an LE scan procedure. This command returns zero if successful and a negative value if an error occurred. This command calls the StartScan(unsigned int BluetoothStackID) function which performs the scan.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Scan.

Possible Return Values

(0) Successfully started the LE Scan Procedure

(-1) Bluetooth Stack ID is Invalid during the StartScan() call

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1) BTPS_ERROR_INVALID_PARAMETER

(-66) BTPS_ERROR_INSUFFICIENT_RESOURCES

(-105) BTPS_ERROR_SCAN_ACTIVE

(-56) BTPS_ERROR_GAP_NOT_INITIALIZED

(-104) BTPS_ERROR_LOCAL_CONTROLLER_DOES_NOT_SUPPORT_LE

(-57) BTPS_ERROR_DEVICE_HCI_ERROR

API Call

GAP_LE_Perform_Scan(BluetoothStackID, stActive, 10, 10, latPublic, fpNoFilter, TRUE, GAP_LE_Event_Callback, 0);

API Prototype

int BTPSAPI GAP_LE_Perform_Scan(unsigned int BluetoothStackID, GAP_LE_Scan_Type_t ScanType, unsigned int ScanInterval, unsigned int ScanWindow, GAP_LE_Address_Type_t LocalAddressType, GAP_LE_Filter_Policy_t FilterPolicy, Boolean_t FilterDuplicates, GAP_LE_Event_Callback_t GAP_LE_Event_Callback, unsigned long CallbackParameter);

Description of API

The GAP_LE_Perform_Scan function is provided to allow the local host the ability to begin an LE scanning procedure. This procedure is similar in concept to the inquiry procedure in Bluetooth BR/EDR in that it can be used to discover devices that have been instructed to advertise. This function will return zero if successful, or a negative return error code if there was an error condition.

StopScanning

Description

The StopScanning command is responsible for stopping an LE scan procedure. This command returns zero if successful and a negative value if an error occurred. This command calls the StopScan(unsigned int BluetoothStackID) function which performs the scan.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of disabling Scanning.

Possible Return Values

(0) Successfully stopped the LE Scan Procedure

(-1) Bluetooth Stack ID is Invalid during the StartScan() call

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1) BTPS_ERROR_INVALID_PARAMETER

(-56) BTPS_ERROR_GAP_NOT_INITIALIZED

(-104) BTPS_ERROR_LOCAL_CONTROLLER_DOES_NOT_SUPPORT_LE

(-57) BTPS_ERROR_DEVICE_HCI_ERROR

API Call

GAP_LE_Cancel_Scan(BluetoothStackID);

API Prototype

int BTPSAPI GAP_LE_Cancel_Scan(unsigned int BluetoothStackID);

Description of API

The GAP_LE_Cancel_Scan function is provided to allow the local host the ability to cancel (stop) an on-going scan procedure. This function will return zero if successful or a negative return error code if there was an error condition.

ConnectLE

Description

The ConnectLE command is responsible for connecting to an LE device. This command returns zero if successful and a negative value if an error occurred. This command calls the ConnectLEDevice(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, Boolean_t UseWhiteList) function using ConnectLEDevice(BluetoothStackID, BD_ADDR, FALSE).

Parameters

The only parameter required is the Bluetooth Address of the remote device. This can easily be found using the StartScanning command if the advertising device is in proximity during the scan.

Command Call Examples

“ConnectLE 001bdc05b617” Attempts to send a connection request to the Bluetooth Device with the BD_ADDR of 001bdc05b617.

“ConnectLE 000275e126FF” Attempts to send a connection request to the Bluetooth Device with the BD_ADDR of 000275e126FF.

Possible Return Values

(0) Successfully Set Pairability Mode

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-116) BTPS_ERROR_RANDOM_ADDRESS_IN_USE

(-111) BTPS_ERROR_CREATE_CONNECTION_OUTSTANDING

(-66) BTPS_ERROR_INSUFFICIENT_RESOURCES

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1) BTPS_ERROR_INVALID_PARAMETER

(-56) BTPS_ERROR_GAP_NOT_INITIALIZED

(-104) BTPS_ERROR_LOCAL_CONTROLLER_DOES_NOT_SUPPORT_LE

(-57) BTPS_ERROR_DEVICE_HCI_ERROR

GAP_LE_ERROR_WHITE_LIST_IN_USE

API Calls

GAP_LE_Create_Connection(BluetoothStackID, 100, 100, Result?fpNoFilter:fpWhiteList, latPublic, Result? &BD_ADDR:NULL, latPublic, &ConnectionParameters, GAP_LE_Event_Callback, 0);

(these two APIs can generally be ignored unless the WhiteList is enabled in the call to ConnectLEDevice)

GAP_LE_Remove_Device_From_White_List(BluetoothStackID, 1, &WhiteListEntry, &WhiteListChanged);

GAP_LE_Add_Device_To_White_List(BluetoothStackID, 1, &WhiteListEntry, &WhiteListChanged);

API Prototypes

*int BTPSAPI GAP_LE_Create_Connection(unsigned int BluetoothStackID, unsigned int ScanInterval, unsigned int ScanWindow, GAP_LE_Filter_Policy_t InitatorFilterPolicy, GAP_LE_Address_Type_t RemoteAddressType, BD_ADDR_t *RemoteDevice, GAP_LE_Address_Type_t LocalAddressType, GAP_LE_Connection_Parameters_t *ConnectionParameters, GAP_LE_Event_Callback_t GAP_LE_Event_Callback, unsigned long CallbackParameter);*

*int BTPSAPI GAP_LE_Remove_Device_From_White_List(unsigned int BluetoothStackID, unsigned int DeviceCount, GAP_LE_White_List_Entry_t *WhiteListEntries, unsigned int *RemovedDeviceCount);*

*int BTPSAPI GAP_LE_Add_Device_To_White_List(unsigned int BluetoothStackID, unsigned int DeviceCount, GAP_LE_White_List_Entry_t *WhiteListEntries, unsigned int *AddedDeviceCount);*

Description of API

The GAP_LE_Create_Connection function is provided to allow the local host the ability to create a connection to a remote device using the Bluetooth LE radio. The connection process is asynchronous in nature and the caller will be notified via the GAP LE event callback function (specified in this function) when the connection completes. This function will return zero if successful, or a negative return error code if there was an error condition. The GAP_LE_Remove_Device_From_White_List function is provided to allow the local host the ability to remove one (or more) devices from the white list maintained by the local device. This function will attempt to delete as many devices as possible (from the specified list) and will return the number of devices deleted. The GAP_LE_Read_White_List_Size function can be used to determine how many devices the local device supports in the white list (simultaneously).

DisconnectLE

Description

The DisconnectLE command is responsible for disconnecting from an LE device. This command returns zero on successful execution and a negative value on all errors. This command requires that a valid Bluetooth Stack ID exists before running.

Parameters

The only parameter required is the Bluetooth Address of the remote device that is connected.

Command Call Examples

“DisconnectLE 001bdc05b617” Attempts to send a disconnection request to the Bluetooth Device with the BD_ADDR of 001bdc05b617.

“DisconnectLE 000275e126FF” Attempts to send a disconnection request to the Bluetooth Device with the BD_ADDR of 000275e126FF.

Possible Return Values

(0) Successfully disconnected remote device (-4) FUNCTION_ERROR (-8) INVALID_STACK_ID_ERROR

API Call

GAP_LE_Disconnect(BluetoothStackID, BD_ADDR);

API Prototype

int BTPSAPI GAP_LE_Disconnect(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR);

API Description

The GAP_LE_Disconnect function provides the ability to disconnect from a remote device. This function will return zero if successful, or a negative return error code if there was an error condition.

PairLE

Description

The PairLE command is provided to allow a mechanism of Pairing (or requesting security if a slave) to the connected device. This command calls the SendPairingRequest(BD_ADDR_t BD_ADDR, Boolean_t ConnectionMaster) function using SendPairingRequest(ConnectionBD_ADDR, LocalDeviceIsMaster).

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of Pairing.

Possible Return Values

(0) Successfully Set Pairability Mode

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1) BTPS_ERROR_INVALID_PARAMETER

(-56) BTPS_ERROR_GAP_NOT_INITIALIZED

(-104) BTPS_ERROR_LOCAL_CONTROLLER_DOES_NOT_SUPPORT_LE

(-66) BTPS_ERROR_INSUFFICIENT_RESOURCES

(-107) BTPS_ERROR_INVALID_DEVICE_ROLE_MODE

API Calls

GAP_LE_Pair_Remote_Device(BluetoothStackID, BD_ADDR, &Capabilities, GAP_LE_Event_Callback, 0);

GAP_LE_Request_Security(BluetoothStackID, BD_ADDR, Capabilities.Bonding_Type, Capabilities.MITM, GAP_LE_Event_Callback, 0);

API Prototypes

*int BTPSAPI GAP_LE_Pair_Remote_Device(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, GAP_LE_Pairing_Capabilities_t *Capabilities, GAP_LE_Event_Callback_t GAP_LE_Event_Callback, unsigned long CallbackParameter);*

int BTPSAPI GAP_LE_Request_Security(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, GAP_LE_Bonding_Type_t Bonding_Type, Boolean_t MITM, GAP_LE_Event_Callback_t GAP_LE_Event_Callback, unsigned long CallbackParameter);

Description of API

The GAP_LE_Pair_Remote_Device function is provided to allow a means to pair with a remote, connected, device. This function accepts the device address of the currently connected device to pair with, followed by the pairing capabilities of the local device. This function also accepts as input the GAP LE event callback information to use during the pairing process. This function returns zero if successful or a negative error code if there was an error. This function can only be issued by the master of the connection (the initiator of the connection). The reason is that a slave can only request a security procedure, it cannot initiate a security procedure. The GAP_LE_Request_Security function is provided to allow a means for a slave device to request that the master (of the connection) perform a pairing operation or re-establishing prior security. This function can only be called by a slave device. The reason for this is that the slave can only request for security to be initiated, it cannot initiate the security process itself. This function returns zero if successful or a negative error code if there was an error.

LEPassKeyResponse

Description

The LEPassKeyResponse command is responsible for issuing a GAP Authentication Response with a Pass Key value specified via the input parameter. This command returns zero on successful execution and a negative value on all errors.

Parameters

The PassKeyResponse command requires one parameter which is the Pass Key used for authenticating the connection. This is a string value which can be up to 6 digits long (with a value between 0 and 999999).

Command Call Examples

"PassKeyResponse 1234" Attempts to set the Pass Key to "1234." "PassKeyResponse 999999" Attempts to set the Pass Key to "999999." This value represents the longest Pass Key value of 6 digits.

Possible Return Values

(0) Successful Pass Key Response

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1) BTPS_ERROR_INVALID_PARAMETER

(-118) BTPS_ERROR_PAIRING_NOT_ACTIVE

(-57) BTPS_ERROR_DEVICE_HCI_ERROR

(-56) BTPS_ERROR_GAP_NOT_INITIALIZED

(-104) BTPS_ERROR_LOCAL_CONTROLLER_DOES_NOT_SUPPORT_LE

(-66) BTPS_ERROR_INSUFFICIENT_RESOURCES

(-107) BTPS_ERROR_INVALID_DEVICE_ROLE_MODE

API Call

GAP_LE_Authentication_Response(BluetoothStackID, CurrentRemoteBD_ADDR, &GAP_LE_Authentication_Response_Information);

API Prototype

*int BTPSAPI GAP_LE_Authentication_Response(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, GAP_LE_Authentication_Response_Information_t *GAP_LE_Authentication_Information);*

Description of API

This function is provided to allow a mechanism for the local device to respond to GAP LE authentication events. This function is used to specify the authentication information for the specified Bluetooth device. This function accepts as input, the Bluetooth protocol stack ID of the Bluetooth device that has requested the authentication action, and the authentication response information (specified by the caller).

LEQueryEncryption

Description

The LEQueryEncryption command is responsible for quering the Encryption Mode for an LE Connection. This command returns zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Query.

Possible Return Values

(0) Successfully Queried Encryption Mode

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1) BTPS_ERROR_INVALID_PARAMETER

(-56) BTPS_ERROR_GAP_NOT_INITIALIZED

(-104) BTPS_ERROR_LOCAL_CONTROLLER_DOES_NOT_SUPPORT_LE

API Call

GAP_LE_Query_Encryption_Mode(BluetoothStackID, ConnectionBD_ADDR, &GAP_Encryption_Mode);

API Prototype

*int BTPSAPI GAP_LE_Query_Encryption_Mode(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, GAP_Encryption_Mode_t *GAP_Encryption_Mode);*

Description of API

This function is provided to allow a means to query the current encryption mode for the LE connection that is specified.

SetPasskey

Description

The SetPasskey command is responsible for querying the Encryption Mode for an LE Connection. This command returns zero on successful execution and a negative value on all errors.

Note

SetPasskey Command works only when you are pairing.

Parameters

The SetPasskey command requires one parameter which is the Pass Key used for authenticating the connection. This is a string value which can be up to 6 digits long (with a value between 0 and 999999).

Command Call Examples

“SetPasskey 0” Attempts to remove the Passkey.

“SetPasskey 1 987654” Attempts to set the Passkey to 987654.

“SetPasskey 1” Attempts to set the Passkey to the default Fixed Passkey value.

Possible Return Values

(0) Successful Pass Key Response

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1) BTPS_ERROR_INVALID_PARAMETER

(-56) BTPS_ERROR_GAP_NOT_INITIALIZED

(-104) BTPS_ERROR_LOCAL_CONTROLLER_DOES_NOT_SUPPORT_LE

API Calls

(Depending on the First Parameter one of these will be chosen)

GAP_LE_Set_Fixed_Passkey(BluetoothStackID, &Passkey);

GAP_LE_Set_Fixed_Passkey(BluetoothStackID, NULL);

API Prototype

*int BTPSAPI GAP_LE_Set_Fixed_Passkey(unsigned int BluetoothStackID, DWord_t *Fixed_Display_Passkey);*

Description of API

This function is provided to allow a means for a fixed passkey to be used whenever the local Bluetooth device is chosen to display a passkey during a pairing operation. This fixed passkey is only used when the local Bluetooth device is chosen to display the passkey, based on the remote I/O Capabilities and the local I/O capabilities.

DiscoverGAPS

Description

The DiscoverGAPS command is provided to allow an easy mechanism to start a service discovery procedure to discover the Generic Access Profile Service on the connected remote device.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the service discovery.

Possible Return Values

(0) Successfully discovered the Generic Access Profile Service.

(-4) Function Error (on failure).

API Call

GDIS_Service_Discovery_Start(BluetoothStackID, ConnectionID, (sizeof(UUID)/sizeof(GATT_UUID_t)), UUID, GDIS_Event_Callback, sdGAPS)

API Prototypes

*int BTPSAPI GDIS_Service_Discovery_Start(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GDIS_Event_Callback_t ServiceDiscoveryCallback, unsigned long ServiceDiscoveryCallbackParameter)*

Description of API

The GDIS_Service_Discover_Start is in an application module called GDIS that is provided to allow an easy way to perform GATT service discovery. This module can and should be modified for the customers use. This function is called to start a service discovery operation by the GDIS module.

GetLocalName

Description

The GetLocalName command is responsible for querying the name of the local Bluetooth Device. This command returns zero on a successful execution and a negative value on all errors. A Bluetooth Stack ID must exist before attempting to call this command.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the Query.

Possible Return Values

(0) Successfully Queried Local Device Name

(-8) INVALID_STACK_ID_ERROR

(-4) FUNCTION_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1) BTPS_ERROR_INVALID_PARAMETER

(-57) BTPS_ERROR_DEVICE_HCI_ERROR

(-65) BTPS_ERROR_INSUFFICIENT_BUFFER_SPACE

API Call

*GAP_Query_Local_Device_Name(BluetoothStackID, 257, (char *)LocalName);*

API Prototype

*int BTPSAPI GAP_Query_Local_Device_Name(unsigned int BluetoothStackID, unsigned int NameBufferLength, char *NameBuffer);*

Description of API

This function is responsible for querying (and reporting) the user friendly name of the local Bluetooth device. The final parameters to this function specify the buffer and buffer length of the buffer that is to receive the local device name. The NameBufferLength parameter should be at least (MAX_NAME_LENGTH+1) to hold the maximum allowable device name (plus a single character to hold the NULL terminator). If this function is successful, this function returns zero, and the buffer that NameBuffer points to will be filled with a NULL terminated ASCII representation of the local device name. If this function returns a negative value, then the local device name was NOT able to be queried (error condition).

SetLocalName

Description

The SetLocalName command is responsible for setting the name of the local Bluetooth Device to a specified name. This command returns zero on a successful execution and a negative value on all errors. A Bluetooth Stack ID must exist before attempting to call this command.

Parameters

One parameter is necessary for this command. The specified device name must be the only parameter (which means there should not be spaces in the name or only the first section of the name will be set).

Command Call Examples

"SetLocalName New_Bluetooth_Device_Name" Attempts to set the Local Device Name to "New_Bluetooth_Device_Name." "SetLocalName New Bluetooth Device Name" Attempts to set the Local Device Name to "New Bluetooth Device Name" but only sets the first parameter, which would make the Local Device Name "New." "SetLocalName MSP430" Attempts to set the Local Device Name to "MSP430."

Possible Return Values

(0) Successfully Set Local Device Name

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-8) INVALID_STACK_ID_ERROR

(-4) FUNCTION_ERROR

(-57) BTPS_ERROR_DEVICE_HCI_ERROR

API Call

GAP_Set_Local_Device_Name(BluetoothStackID, TempParam->Params[0].strParam);

API Prototype

*int BTPSAPI GAP_Set_Local_Device_Name(unsigned int BluetoothStackID, char *Name);*

Description of API

This function is provided to allow the changing of the device name of the local Bluetooth device. The Name parameter must be a pointer to a NULL terminated ASCII string of at most MAX_NAME_LENGTH (not counting the trailing NULL terminator). This function will return zero if the local device name was successfully changed, or a negative return error code if there was an error condition.

GetRemoteName

Description

The GetRemoteName command is responsible for querying the Bluetooth Device Name of a Remote Device. This command returns zero on a successful execution and a negative value on all errors. The command requires that a valid Bluetooth Stack ID exists before running and it should be called after using the Inquiry command. The DisplayInquiryList command would be useful in this situation to find which Remote Device goes with which Inquiry Index.

Parameters

The GetRemoteName command requires one parameter which is the Inquiry Index of the Remote Bluetooth Device. This value can be found after an Inquiry or displayed when the command DisplayInquiryList is used. Command Call Examples "GetRemoteName 5" Attempts to query the Device Name for the Remote Device that is at the fifth Inquiry Index. "GetRemoteName 8" Attempts to query the Device Name for the Remote Device that is at the eighth Inquiry Index.

Possible Return Values

(0) Successfully Queried Remote Name

(-6) INVALID_PARAMETERS_ERROR

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1) BTPS_ERROR_INVALID_PARAMETER

(-59) BTPS_ERROR_ADDING_CALLBACK_INFORMATION

(-57) BTPS_ERROR_DEVICE_HCI_ERROR

API Call

GAP_Query_Remote_Device_Name(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam – 1)], GAP_Event_Callback, (unsigned long)0);

API Prototype

int BTPSAPI GAP_Query_Remote_Device_Name(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, GAP_Event_Callback_t GAP_Event_Callback, unsigned long CallbackParameter);

Description of API

This function is provided to allow a mechanism to query the user-friendly Bluetooth device name of the specified remote Bluetooth device. This function accepts as input the Bluetooth device address of the remote Bluetooth device to query the name of and the GAP event callback information that is to be used when the remote device name process has completed. This function returns zero if successful, or a negative return error code if the remote name request was unable to be submitted. If this function returns success, then the caller will be notified via the specified callback when the remote name information has been determined (or there was an error). This function cannot be used to determine the user-friendly name of the local Bluetooth device. The GAP_Query_Local_Name function should be used to query the user-friendly name of the local Bluetooth device. Because this function is asynchronous in nature (specifying a remote device address), this function will notify the caller of the result via the specified callback. The caller is free to cancel the remote name request at any time by issuing the GAP_Cancel_Query_Remote_Name function and specifying the Bluetooth device address of the Bluetooth device that was specified in the original call to this function. It should be noted that when the callback is cancelled, the operation is attempted to be cancelled and the callback is cancelled (i.e. the GAP module still might perform the remote name request, but no callback is ever issued).

LEUserConfirmationResponse

Description

The LEUserConfirmationResponse command is responsible for issuing a GAP LE Authentication Response with a User Confirmation value specified via the input parameter. This function returns zero on successful execution and a negative value on all errors.

Parameters

This command requires one parameter which indicates if confirmation is accepted or not. 0 = decline, 1 = accept.

Command Call Examples

“LEUserConfirmationResponse 0” Attempts to Response with a decline value.

“LEUserConfirmationResponse 1” Attempts to Response with a accept value.

Possible Return Values

(0) Success.

(-4) FUNCTION_ERROR.

(-6) INVALID_PARAMETERS_ERROR.

(-1) BTPS_ERROR_INVALID_PARAMETER.

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID.

(-56) BTPS_ERROR_GAP_NOT_INITIALIZED.

(-57) BTPS_ERROR_DEVICE_HCI_ERROR.

(-66) BTPS_ERROR_INSUFFICIENT_RESOURCES.

(-98) BTPS_ERROR_DEVICE_NOT_CONNECTED.

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE.

(-104) BTPS_ERROR_LOCAL_CONTROLLER_DOES_NOT_SUPPORT_LE.

(-107) BTPS_ERROR_INVALID_DEVICE_ROLE_MODE.

(-118) BTPS_ERROR_PAIRING_NOT_ACTIVE.

(-119) BTPS_ERROR_INVALID_STATE.

(-120) BTPS_ERROR_FEATURE_NOT_CURRENTLY_ACTIVE.

(-122) BTPS_ERROR_NUMERIC_COMPARISON_FAILED.

API Call

GAP_LE_Authentication_Response(BluetoothStackID, CurrentLERemoteBD_ADDR, &GAP_LE_Authentication_Response_Information)

API Prototype

*int BTPSAPI GAP_LE_Authentication_Response(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, GAP_LE_Authentication_Response_Information_t *GAP_LE_Authentication_Information)*

Description of API

The following function is provided to allow a mechanism for the local device to respond to GAP LE authentication events. This function is used to set the authentication information for the specified Bluetooth device. This function accepts as input, the Bluetooth protocol stack ID followed by the remote Bluetooth device address that is currently executing a pairing/authentication process, followed by the authentication response information.This function returns zero if successful, or a negative return error code if there was an error.

EnableSCOnly

Description

The EnableSCOnly command enables LE Secure Connections (SC) only mode. In case this mode is enabled, pairing request from peers that support legacy pairing only will be rejected. Please note that in case this mode is enabled, the SC flag in the LE_Parameters must be set to TRUE. This function returns zero on successful execution and a negative value on all errors.

Parameters

This command requires one parameter which indicates if Secure connections only mode is set or not. 0 = SC Only mode is off, 1 = SC Only mode is on.

Command Call Examples

“EnableSCOnly 0” Disable Secure connections only mode.

“EnableSCOnly 1” Enable Secure connections only mode.

Possible Return Values

(0) Success.

(-4) FUNCTION_ERROR.

(-6) INVALID_PARAMETERS_ERROR.

(-8) INVALID_STACK_ID_ERROR.

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID.

(-56) BTPS_ERROR_GAP_NOT_INITIALIZED.

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE.

(-104) BTPS_ERROR_LOCAL_CONTROLLER_DOES_NOT_SUPPORT_LE.

(-120) BTPS_ERROR_FEATURE_NOT_CURRENTLY_ACTIVE.

API Call

GAP_LE_SC_Only_Mode(BluetoothStackID, EnableSCOnly)

API Prototype

int BTPSAPI GAP_LE_SC_Only_Mode(unsigned int BluetoothStackID, Boolean_t EnableSCOnly)

Description of API

The following function is provided to allow a configuration of LE Secure Connecions only mode. The upper layer will use this function before the beginning of LE SC pairing, in case it asks to reject a device that supports only legacy pairing. This mode should be used when it is more important for a device to have high security than it is for it to maintain backwards compatibility with devices that do not support SC. This function accepts as parameters the Bluetooth stack ID of the Bluetooth device,and a boolean EnableSCOnly that enable or disable the SC only mode. This function should be used ones, before the first pairing process. This function returns zero if successful or a negative error code.

RegenerateP256LocalKeys

Description

The following function allows the user to generate new P256 private and local keys. This function shall NOT be used in the middle of a pairing process. It is relevant for LE Secure Conenctions pairing only! This function returns zero on successful execution and a negative value on all errors.

Parameters

No parameters are necessary.

Command Call Examples

“RegenerateP256LocalKeys” Attempts to generate new P256 private and local keys.

Possible Return Values

(0) Success.

(-4) FUNCTION_ERROR.

(-8) INVALID_STACK_ID_ERROR.

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID.

(-56) BTPS_ERROR_GAP_NOT_INITIALIZED.

(-104) BTPS_ERROR_LOCAL_CONTROLLER_DOES_NOT_SUPPORT_LE.

(-117) BTPS_ERROR_PAIRING_ACTIVE.

(-120) BTPS_ERROR_FEATURE_NOT_CURRENTLY_ACTIVE.

API Call

GAP_LE_SC_Regenerate_P256_Local_Keys(BluetoothStackID)

API Prototype

int BTPSAPI GAP_LE_SC_Regenerate_P256_Local_Keys(unsigned int BluetoothStackID)

Description of API

The following function is provided to allow a regeneration of the P-256 private and local puclic keys. This function is relevant only in case of LE SC pairing. This function accepts as parameters the Bluetooth stack ID of the Bluetooth device. This functions shall NOT be used while performing pairing. This function returns zero if successful or a negative error code.

SCGenerateOOBLocalParams

Description

In order to be able to perform LE SC pairing in OOB method we need to generate local random and confirmation values before the pairing process starts. The following function allows the user to generate OOB local parameters. This function shall NOT be used in the middle of a pairing process. It is relevant for LE SC pairing only! This function returns zero on successful execution and a negative value on all errors.

Parameters

No parameters are necessary.

Command Call Examples

“SCGenerateOOBLocalParams” Attempts to generate local random and confirmation values before the pairing process starts.

Possible Return Values

(0) Success.

(-4) FUNCTION_ERROR.

(-8) INVALID_STACK_ID_ERROR.

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID.

(-56) BTPS_ERROR_GAP_NOT_INITIALIZED.

(-104) BTPS_ERROR_LOCAL_CONTROLLER_DOES_NOT_SUPPORT_LE.

(-117) BTPS_ERROR_PAIRING_ACTIVE.

(-120) BTPS_ERROR_FEATURE_NOT_CURRENTLY_ACTIVE.

API Call

GAP_LE_SC_OOB_Generate_Parameters(BluetoothStackID, &OOBLocalRandom, &OOBLocalConfirmation)

API Prototype

*int BTPSAPI GAP_LE_SC_OOB_Generate_Parameters(unsigned int BluetoothStackID, SM_Random_Value_t *OOB_Local_Rand_Result, SM_Confirm_Value_t *OOB_Local_Confirm_Result)*

Description of API

The following function is provided to allow the use of LE Secure Connections (SC) pairing in Out Of Band (OOB) association method. The upper layer will use this function to generate the the local OOB random value, and

OOB confirmation value (ra/rb and Ca/Cb) as defined in the Bluetooth specification. This function accepts as parameters the Bluetooth stack ID of the Bluetooth device, and pointers to buffers that will recieve the generated local OOB random, and OOB confirmation values. This function returns zero if successful or a negative error code.

SetLocalAppearence

Description

The SetLocalAppearence command is provided to set the local device appearance that is exposed by the GAP Service (GAPS).

Parameters

The SetLocalAppearence command requires one parameter which is the Local Device Appearance you wish to be set.

Possible Return Values

(0) Success.

(-4) Function error (on failure).

API Call

GAPS_Set_Device_Appearance(BluetoothStackID, GAPSInstanceID, Appearance)

API Prototype

int BTPSAPI GAPS_Set_Device_Appearance(unsigned int BluetoothStackID, unsigned int InstanceID, Word_t DeviceAppearance);

Description of API

This function allows a mechanism of setting the local device appearance that is exposed as part of the GAP Service API (GAPS).

GetLocalAppearence

Description

The GetLocalAppearence command is provided to read the local device appearance that is exposed by the GAP Service (GAPS).

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Success.

(-4) Function error (on failure).

API Call

GAPS_Query_Device_Appearance(BluetoothStackID, GAPSInstanceID, &Appearance)

API Prototype

*int BTPSAPI GAPS_Query_Device_Appearance(unsigned int BluetoothStackID, unsigned int InstanceID, Word_t *DeviceAppearance)*

Description of API

This function allows a mechanism of reading the local device appearance that is exposed as part of the GAP Service API (GAPS).

GetRemoteAppearence

Description

The GetRemoteAppearence command is provided to read the device appearance from the connected remote device that is exposed as part of the GAP Service. The GAP Service on the remote device must have already been discovered using the DiscoverGAPS command.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome

Possible Return Values

(0) Success.

(-4) Function error (on failure).

API Call

GATT_Read_Value_Request(BluetoothStackID, ConnectionID, DeviceInfo->GAPSClientInfo.DeviceAppearanceHandle, GATT_ClientEventCallback_GAPS, (unsigned long)DeviceInfo->GAPSClientInfo.DeviceAppearanceHandle)

API Prototype

int BTPSAPI GATT_Read_Value_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, GATT_Client_Event_Callback_t ClientEventCallback, unsigned long CallbackParameter)

Description of API

This function allows a mechanism of reading an attribute from a connected device.

SPPLE Commands
DiscoverSPPLE

Description

The following function is responsible for performing a SPPLE Service Discovery Operation. This function will return zero on successful execution and a negative value on errors.

Parameters

The only parameter required is the Bluetooth Address of the remote device that is connected.

Command Call Examples

“DiscoverSPPLE 001bdc05b617” Attempts to discover services of the Bluetooth Device with the BD_ADDR of 001bdc05b617.

“DiscoverSPPLE 000275e126FF” Attempts to discover services of the Bluetooth Device with the BD_ADDR of 000275e126FF.

Possible Return Values

(0) Successfully started a SPP LE Service Discovery.

(-4) Function Error (on failure).

API Call

GDIS_Service_Discovery_Start(BluetoothStackID, ConnectionID, (sizeof(UUID)/sizeof(GATT_UUID_t)), UUID, GDIS_Event_Callback, 0)

API Prototype

*int BTPSAPI GDIS_Service_Discovery_Start(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GDIS_Event_Callback_t ServiceDiscoveryCallback, unsigned long ServiceDiscoveryCallbackParameter)*

Description of API

The GDIS_Service_Discover_Start is in an application module called GDIS that is provided to allow an easy way to perform GATT service discovery. This function is called to start a service discovery operation by the GDIS module.

RegisterSPPLE

Description

The following function is responsible for registering a SPPLE Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of registering a SPPLE Service.

Possible Return Values

(0) Successfully registered a SPPLE Service.

(-4) Function Error (on failure).

API Call

*GATT_Register_Service(BluetoothStackID, SPPLE_SERVICE_FLAGS, SPPLE_SERVICE_ATTRIBUTE_COUNT, (GATT_Service_Attribute_Entry_t *)SPPLE_Service, &ServiceHandleGroup, GATT_ServerEventCallback, 0)*

API Prototype

*int BTPSAPI GATT_Register_Service(unsigned int BluetoothStackID, Byte_t ServiceFlags, unsigned int NumberOfServiceAttributeEntries, GATT_Service_Attribute_Entry_t *ServiceTable, GATT_Attribute_Handle_Group_t *ServiceHandleGroupResult, GATT_Server_Event_Callback_t ServerEventCallback, unsigned long CallbackParameter)*

Description of API

The following function is provided to allow a means to add a GATT Service to the local GATT Database. The first parameter is Bluetooth stack ID of the Bluetooth Device. The second parameter is a bit mask field that specifies the type of service being registered, which must be non-zero (i.e. at least one bit must be set). The third parameter is the number of entries in the service attribute array that is pointed to by the fourth parameter. The fourth parameter is an array that contains the attributes for the service being registered. The next parameter is a pointer to a buffer that will store the attribute handle range of the registered service. The final two parameters specify the GATT server callback and callback parameter that will be used whenever a client request to the GATT server cannot be satisified internally by the local GATT module. This function will return a positive non-zero service ID if successful, or a negative return error code if there was an error. If this function returns success then the ServiceHandleGroupResult buffer will contain the service's attribute handle range.

LESend

Description

The following function is responsible for sending a number of characters to a remote device to which a connection exists. The function receives a parameter that indicates the number of byte to be transferred. This function will return zero on successful execution and a negative value on errors. Depending what the device role for SPPLE is, server or client, the API function that is called is either a GATT_Handle_Value_Notification or a GATT_Write_Without_Response_Request; which notifies the receiving credit characteristic or sends a write with out response packet to the transmission credit characteristic respectively.

Parameters

Lesend requires two parameters. The first is the remote Bluetooth address of the device you are sending to. The second is the number of bytes to send. This value has to be greater than 10.

Command Call Examples

"LeSend 0017E7FEFD7C 100" Attempts to send 100 bytes of data to 0017E7FEFD7C.

"LeSend B8FFFEAF1CAD 25" Attempts to send 25 bytes of data to B8FFFEAF1CAD.

Possible Return Values

(0) Successfully Sent Data

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

API Call

GATT_Handle_Value_Notification(BluetoothStackID, SPPLEServiceID, ConnectionID, SPPLE_TX_CHARACTERISTIC_ATTRIBUTE_OFFSET, (Word_t)DataCount, SPPLEBuffer)

OR

GATT_Write_Without_Response_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.Rx_Characteristic, (Word_t)DataCount, SPPLEBuffer)

API Prototype

*int BTPSAPI GATT_Handle_Value_Notification(unsigned int BluetoothStackID, unsigned int ServiceID, unsigned int ConnectionID, Word_t AttributeOffset, Word_t AttributeValueLength, Byte_t *AttributeValue)*

OR

*int BTPSAPI GATT_Write_Without_Response_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue)*

Description of API

The first of these API functions allows a means of sending a Handle/Value notification to a remote GATT client. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack. The second parameter is the service ID of the service that is sending the Handle/Value notification. The third parameter specifies the connection ID of the connection to send the Handle/Value notification to. The fourth parameter specifies the offset in the service table (registered via the call to the GATT_Register_Service() function) of the attribute that is being notified. The fifth parameter is the length (in bytes) of the attribute value that is being notified. The sixth parameter is a pointer to the actual attribute value to notify. This function will return a non-negative value that represents the actual length of the attribute value that was notified, or a negative return error code if there was an error.

The second of these API functions is provided to allow a means of performing a write without response request to remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write, followed by the length of the value data to write (in bytes), followed by the actual value to write. This function will return the number of bytes written on success or a negative error code.

ConfigureSPPLE

Description

The following function is responsible to configure a SPPLE Service on a remote device. This function will return zero on successful execution and a negative value on errors. The following function enables notifications of the proper characteristics based on a specified handle; depending what the device role for SPPLE is, server or client, the API function that is called is either a GATT_Handle_Value_Notification or a GATT_Write_Without_Response_Request; which notifies the receiving credit characteristic or sends a write with out response packet to the transmission credit characteristic respectively.

Parameters

The only parameter required is the Bluetooth Address of the remote device that is connected.

Command Call Examples

“ConfigureSPPLE 001bdc05b617” Attempts to configure services of the Bluetooth Device with the BD_ADDR of 001bdc05b617.

“ConfigureSPPLE 000275e126FF” Attempts to configure services of the Bluetooth Device with the BD_ADDR of 000275e126FF.

Possible Return Values

(0) Successfully configured a SPPLE Service.

(-4) Function Error (on failure).

API Call

GATT_Write_Request(BluetoothStackID, ConnectionID, ClientConfigurationHandle, sizeof(Buffer), &Buffer, ClientEventCallback, 0)

AND

*GATT_Handle_Value_Notification(BluetoothStackID, SPPLEServiceID, ConnectionID, SPPLE_RX_CREDITS_CHARACTERISTIC_ATTRIBUTE_OFFSET, WORD_SIZE, (Byte_t *)&Credits)*

OR

GATT_Write_Without_Response_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.Tx_Credit_Characteristic, WORD_SIZE, &Credits)

API Prototype

*int BTPSAPI GATT_Write_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue, GATT_Client_Event_Callback_t ClientEventCallback, unsigned long CallbackParameter)*

AND

*int BTPSAPI GATT_Handle_Value_Notification(unsigned int BluetoothStackID, unsigned int ServiceID, unsigned int ConnectionID, Word_t AttributeOffset, Word_t AttributeValueLength, Byte_t *AttributeValue)*

OR

*int BTPSAPI GATT_Write_Without_Response_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue)*

Description of API

The first of these API functions is provided to allow a means of performing a write request to a remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write the value of, followed by the length of the value (in bytes), followed by the the actual value data to write. The final two parameters specify the GATT client event callback function and callback parameter (respectively) that will be called when a response is received from the remote device. This function will return the positive, non-zero, Transaction ID of the request or a negative error code.

The second of these API functions allows a means of sending a Handle/Value notification to a remote GATT client. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack. The second parameter is the service ID of the service that is sending the Handle/Value notification. The third parameter specifies the connection ID of the connection to send the Handle/Value notification to. The fourth parameter specifies the offset in the service table (registered via the call to the GATT_Register_Service() function) of the attribute that is being notified. The fifth parameter is the length (in bytes) of the attribute value that is being notified. The sixth parameter is a pointer to the actual attribute value to notify. This function will return a non-negative value that represents the actual length of the attribute value that was notified, or a negative return error code if there was an error.

The third of these API functions is provided to allow a means of performing a write without response request to remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the

attribute to write, followed by the length of the value data to write (in bytes), followed by the actual value to write. This function will return the number of bytes written on success or a negative error code.

LERead

Description

The following function is responsible for reading data sent by a remote device to which a connection exists. This function will return zero on successful execution and a negative value on errors. Depending what the device role for SPPLE is, server or client, the API function that is called is either a GATT_Handle_Value_Notification or a GATT_Write_Without_Response_Request; which notifies the receiving credit characteristic or sends a write with out response packet to the transmission credit characteristic respectively.

Parameters

The only parameter required is the Bluetooth Address of the remote device that is connected.

Command Call Examples

“LeRead 001bdc05b617” Attempts to read data of the Bluetooth Device with the BD_ADDR of 001bdc05b617.

“LeRead 000275e126FF” Attempts to read data of the Bluetooth Device with the BD_ADDR of 000275e126FF.

Possible Return Values

(0) Successfully Read Data

(-6) INVALID_PARAMETERS_ERROR

(-1) BTPS_ERROR_INVALID_PARAMETER

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-67) BTPS_ERROR_RFCOMM_NOT_INITIALIZED

(-85) BTPS_ERROR_SPP_NOT_INITIALIZED

(-86) BTPS_ERROR_SPP_PORT_NOT_OPENED

API Call

*GATT_Handle_Value_Notification(BluetoothStackID, SPPLEServiceID, ConnectionID, SPPLE_RX_CREDITS_CHARACTERISTIC_ATTRIBUTE_OFFSET, WORD_SIZE, (Byte_t *)&Credits)*

OR

GATT_Write_Without_Response_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.Tx_Credit_Characteristic, WORD_SIZE, &Credits)

API Prototype

*int BTPSAPI GATT_Handle_Value_Notification(unsigned int BluetoothStackID, unsigned int ServiceID, unsigned int ConnectionID, Word_t AttributeOffset, Word_t AttributeValueLength, Byte_t *AttributeValue)*

OR

*int BTPSAPI GATT_Write_Without_Response_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue)*

Description of API

The first of these API functions allows a means of sending a Handle/Value notification to a remote GATT client. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack. The second parameter is the service ID of the service that is sending the Handle/Value notification. The third parameter specifies the connection ID of the connection to send the Handle/Value notification to. The fourth parameter specifies the offset in the service table (registered via the call to the GATT_Register_Service() function) of the attribute that is being notified. The fifth parameter is the length (in bytes) of the attribute value that is being notified. The sixth parameter is a pointer to the actual attribute value to notify. This function will return a non-negative value that represents the actual length of the attribute value that was notified, or a negative return error code if there was an error.

The second of these API functions is provided to allow a means of performing a write without response request to remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write, followed by the length of the value data to write (in bytes), followed by the actual value to write. This function will return the number of bytes written on success or a negative error code.

Loopback

Description

The Loopback command is responsible for setting the application state to support loopback mode. This command will return zero on successful execution and a negative value on errors.

Parameters

This command requires one parameter which indicates if loopback should be supported. 0 = loopback not active, 1 = loopback active.

Command Call Examples

"Loopback 0" sets loopback support to inactive.

"loopback 1" sets loopback support to active.

Possible Return Values

(0) Successfully set loopback support.

(-6) INVALID_PARAMETERS_ERROR

DisplayRawModeData

Description

The following function is responsible for setting the application state to support displaying Raw Data. This function will return zero on successful execution and a negative value on errors.

Parameters

This command accepts one parameter which indicates if displaying raw data mode should be supported. 0 = Display Raw Data Mode inactive, 1 = Display Raw Data active.

Command Call Examples

"DisplayRawModeData 0" sets Display Raw Mode support inactive.

"DisplayRawModeData 1" sets Display Raw Mode support active.

Possible Return Values

(0) Successfully sets Display Raw Data Mode support.

(-6) INVALID_PARAMETERS_ERROR

AutomaticReadMode

Description

The AutomaticReadMode command is responsible for setting the application state to support Automatically reading all data that is received through SPP. This function will return zero on successful execution and a negative value on errors.

Parameters

This command accepts one parameter which indicates if automatic read mode should be supported. 0 = Automatic Read Mode inactive, 1 = Automatic Read Mode active.

Command Call Examples

"AutomaticReadMode 0" sets Automatic Read Mode support to inactive.

"AutomaticReadMode 1" sets Automatic Read Mode support to active.

Possible Return Values

(0) Successfully set Automatic Read Mode support.

(-6) INVALID_PARAMETERS_ERROR

13 SPPDMMulti Demo Guide

13.1 Demo Overview

These instructions can be used to run this demo on the STM32 and MSP432 Platforms.

This application is based on the SPPLEDemo application and demonstrates how to utilize the chosen features from specification 4.1 and specification 4.2:

specification 4.1 Features:

- LE Dual Mode Topology – connect in classic Bluetooth and Low Energy Bluetooth to the same device.
- LE Link Layer Topology – support multiple simultaneous roles, i.e., Peripheral and Central.
- L2CAP LE Connection Oriented – support LE Credit Based Flow Control Mode.
- LE Ping – used to verify presence of the remote Link Layer. In addition, can be used to verify message integrity in the LE ACL by forcing the remote device to send an LE ACL packet that contains a valid MIC.
- LE Low Duty Cycle Advertising – designed for case where reconnection with a specific device is required, but time is not of the essence or it is not known if the central device is in range or not.
- 32-bit UUID – register services with aliases that are represented as 32-bit values instead of 128-bit UUIDs specification 4.2 Feature.
- LE Secure Connections – enhance the LE Security Manager with public key Elliptic Curve Diffie-Hellman (ECDH) key generation.

The SPPLEDemo application whom this application is based on, Demonstrates a BR/EDR SPP based application as well as a custom application, SPPLE, over Bluetooth LE that is similar in functionality to the BR/EDR appliacation. The SPPLE Profile is similar to the SPP profile except that it uses LE transport compared to BR/EDR transport in the SPP profile.

The SPP profile emulates serial cable connections. There are two roles defined in this profile. The first is the server that has the SPPLE service running on it and has open an server port. The client is a device that connects to the server. Both of these devices can then exchange data with each other.

This document talks about the SPPDMMulti application in details.

To read more about the single connection version of SPPLE refer to this document SPPLE profile.

The application allows the user to use a console to use Bluetooth Low Energy (BLE) to establish connection between two BLE devices, send Bluetooth commands and exchange data over BLE.

It is recommended that the user visits the kit setup Getting Started Guide for STM32 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed. Look at the Device manager for Communications Port (COM x) under Ports (COM & LPT). Attach a terminal program like PuTTY to the serial port (COM x) for the board, x means which COM is open for Communications Port in Device Manager. The serial parameter to use is 115200 Baud rate. Once connected, reset the device using Reset button and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands. This device will become the server.

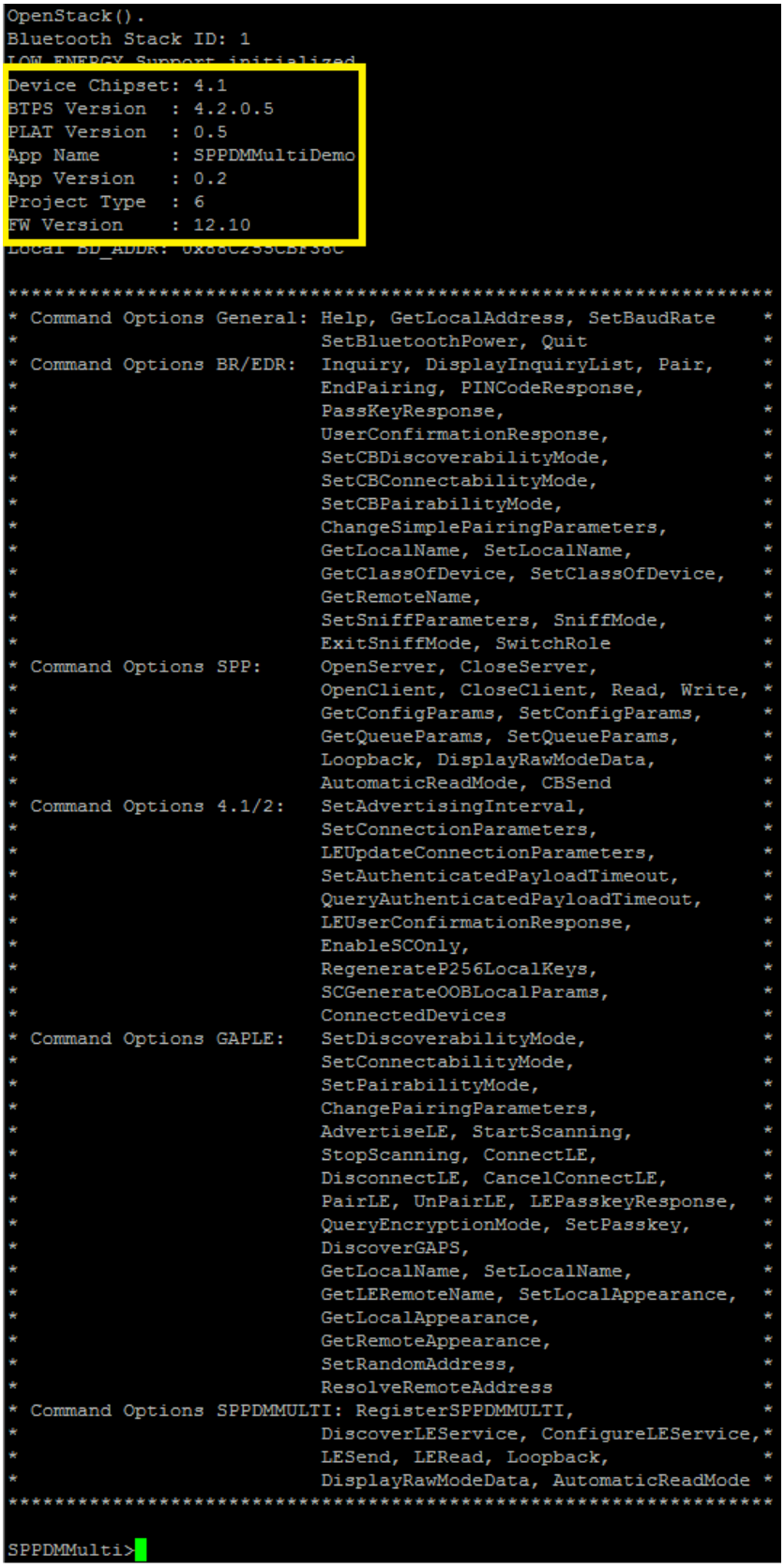

Figure 13-1. SPPDMMulti Demo Start

Note

The yellow square holds specific information on the firmware and software version for future use. Now use the second board and follow the same steps performed before when running the Bluetooth code on the first board. The second device that is connected to the computer will be the client.

13.2 Demo Application

This section provides a description on how to use the demo application to connect two or more configured boards and communicate over bluetooth (BR/EDR and LE). We will setup one of the boards as a central device over LE and Master over BR/EDR and the other board as a peripheral device over LE and Slave over BR/EDR. We will then initiate the connections from the first device to the other. Once connected, we will show how to use the new features over the connection.

Establish Connection
Connection over LE

Peripheral device initialization and advertising

1. We will setup the first board as a Peripheral device. Note the bluetooth address of the server; we will later use this to initiate a connection from the client

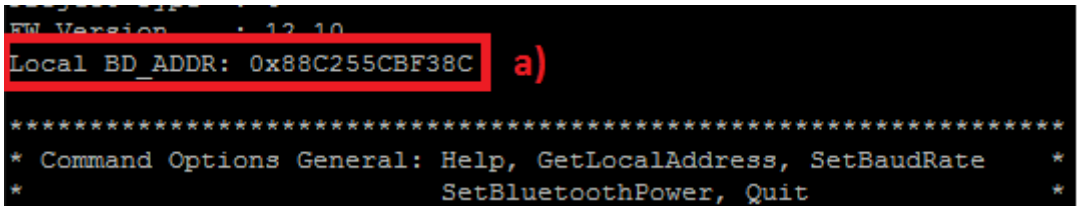

Figure 13-2. SPPDMMulti Demo Local BT Address

2. The first command we need is RegisterSPPDMMULTI, this command will register the SPP-LE service and initiate the Server, type RegisterSPPDMMULTI.

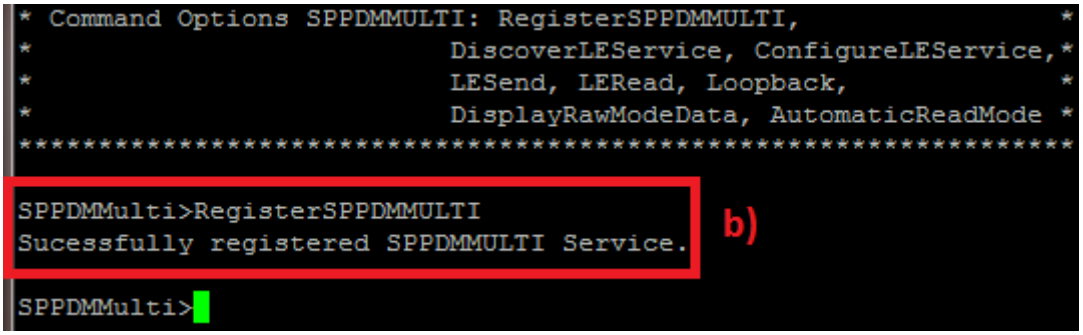

Figure 13-3. SPPDMMulti Demo Register SPPDMMulti

Note

The following steps describe how to configure the Advertising operation, you can change any value that you want and jump to step g to start advertising with the changed values or the default values for the values that weren't changed and follow the instructions.

The default values are:

- Discoverability Mode
- General Discover-able Connectability Mode
- Connectable Own Address Type
- Public address Advertising Intervals
- Minimum interval = 100, Maximum interval - 200

3. In order to configure the Discoverability mode, we need to use the SetDiscoverabilityMode command and the mode that can be:

   0 = Non Discoverable, 1 = Limited Discoverable, 2 = General Discoverable

For this example we will use General Discoverable mode (The same as in the default values).

```
SPPDMMulti>SetDiscoverabilityMode
Usage: SetDiscoverabilityMode [Mode(0 = Non Discoverable, 1 = Limited Discoverable, 2 = General Discoverable)].
Function Error.

SPPDMMulti>SetDiscoverabilityMode 2
Discoverability Mode 2: General Discoverable.

SPPDMMulti>
```

c)

Figure 13-4. SPPDMMulti Demo Set Discoverability Mode

4. In order to configure the Connectability mode, we need to use the SetConnectabilityMode command and the mode that can be:

   0 = Non Conectable, 1 = Connectable, 2 = Direct Connectable, 3 = Low Duty Cycle Direct Connectable.

   For this example we will use Low Duty Cycle Direct Connectable which is a new feature from specification 4.1.

```
SPPDMMulti>SetConnectabilityMode
Usage: SetConnectabilityMode [(0 = Non Conectable, 1 = Connectable,
                               2 = Direct Connectable, 3 = Low Duty Cycle Direct Connectable)].
Function Error.

SPPDMMulti>SetConnectabilityMode 3
Connectability Mode 3: Low Duty Cycle Direct Connectable.

SPPDMMulti>
```

d)

Figure 13-5. SPPDMMulti Demo Set Connect Mode

5. In order to configure the Advertising Intervals, we need to use the SetAdvertisingInterval command and the minimum and maximum intervals that their range can be between 20..10240 in ms

Note

When using High Duty Cycle Direct Connectable this command won't have any affect.

```
SPPDMMulti>SetAdvertisingInterval
Usage: SetAdvertisingInterval.
       [Advertising_Interval_Min] [Advertising_Interval_Max] (Range: 20..10240 in ms)
Function Error.

SPPDMMulti>SetAdvertisingInterval 100 200
SetAdvertisingInterval Success, Parameters stored.
In order to use the new parameters use AdvertiseLE function

SPPDMMulti>
```

e)

Figure 13-6. SPPDMMulti Demo Set Advertising Interval

6. In order to configure the Own Address Type, we need to use the SetRandomAddress command and the type that can be:

   0 = Static, 1 = Resolvable, 2 = Non-Resolvable

   For this example we will use Static random address type.

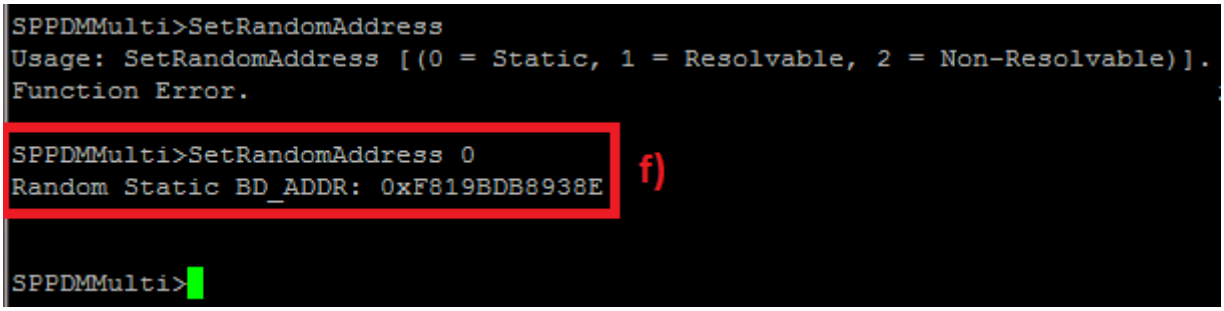

Figure 13-7. SPPDMMulti Demo Set Random Address

7. Now, after we configured the different values we can use the AdvertiseLE command in order to start advertising.

   The AdvertiseLE command has multiple options to advertise:

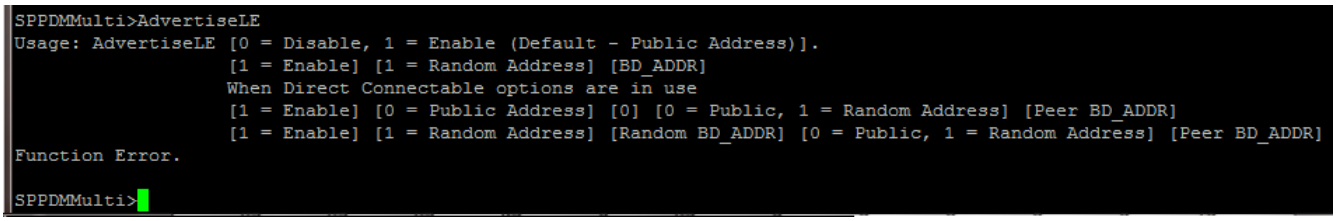

Figure 13-8. SPPDMMulti Demo AdvertiseLE Command

1. Option 1: Disable advertising

   AdvertiseLE 0
2. Option 2: Advertising without any Direct Connectable mode and with own address type public

   AdvertiseLE 1 or AdvertiseLE 1 0
3. Option 3: Advertising without any Direct Connectable mode and with own address type Random

   AdvertiseLE 1 1 [Own Random BD_ADDR]
4. Option 4: Advertising with any Direct Connectable mode and with own address type public and peer type public

   AdvertiseLE 1 0 0 0 [Peer Public BD_ADDR]
5. Option 5: Advertising with any Direct Connectable mode and with own address type Random and peer type public

   AdvertiseLE 1 1 [Own Random BD_ADDR] 0 [Peer Public BD_ADDR]
6. Option 6: Advertising with any Direct Connectable mode and with own address type Random and peer type public

   AdvertiseLE 1 1 [Own Random BD_ADDR] 1 [Peer Random BD_ADDR]

For this example we will use option 5, The device is on Low Duty Cycle Direct Connectable, Own Address type - Random address and the peer device use Public address.

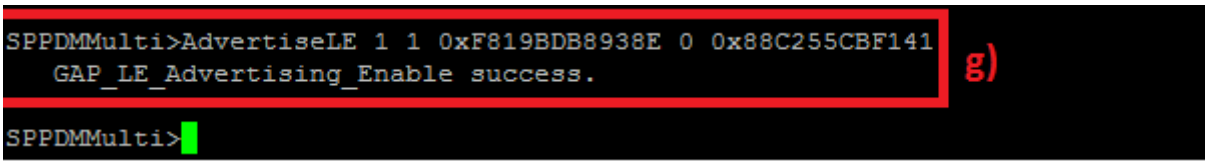

Figure 13-9. SPPDMMulti Demo Enable BLE Advertising

Now the peripheral is advertising and waiting for connection request from the central device.

Central device connection creation and configuration

The Central can initiate the connection request and when the peripheral will advertise the connection will be created.

[Steps 1 and 2 are optional if you already know the Bluetooth address of the device that you want to connect to]

1. The Central device can try to find which LE devices are in the vicinity using the command: StartScanning 0.

Note

When using Public address as Own address type, you can only use StartScanning 0

2. Once you have found the device, you can stop scanning by using the command: StopScanning

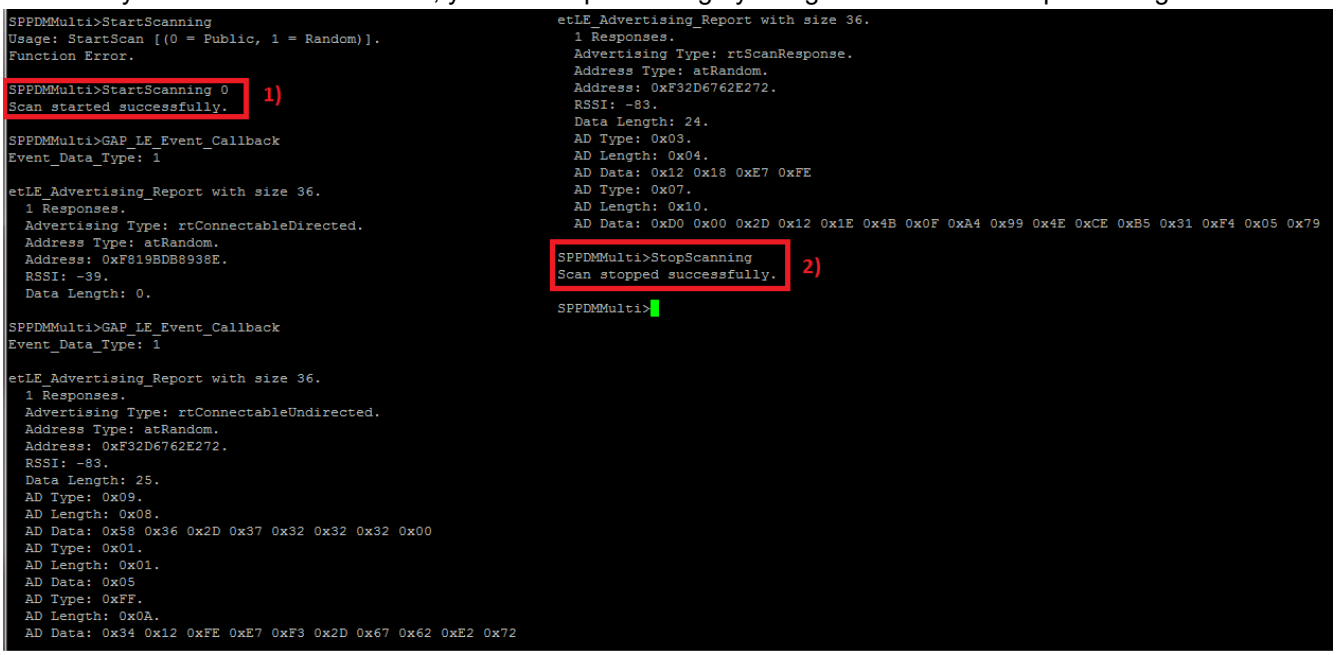

Figure 13-10. SPPDMMulti Demo Scanning Commands

1. Retrieve the Bluetooth address of the first board that was configured as a peripheral.
2. Issue a ConnectLE command in the central terminal in order to start the connection procedure.

The ConnectLE command has multiple options to connect:

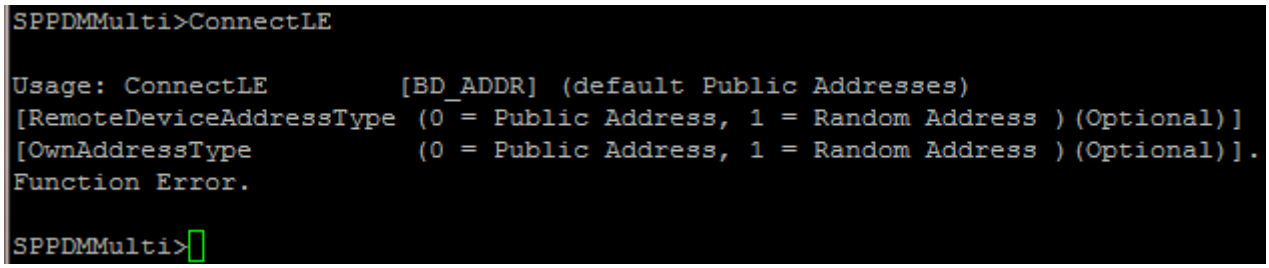

Figure 13-11. SPPDMMulti Demo Connect LE Command

- Option 1: Connect to Remote device using public address type and own address type is also public.

  ConnectLE [Peer Public BD_ADDR]

  ConnectLE [Peer Public BD_ADDR] 0

  ConnectLE [Peer Public BD_ADDR] 0 0
- Option 2: Connect to Remote device using random address type and own address type is public.

  ConnectLE [Peer Random BD_ADDR] 1

  ConnectLE [Peer Random BD_ADDR] 1
- Option 3: Connect to Remote device using random address type and own address type is random.

ConnectLE [Peer Random BD_ADDR] 1 1

- Option 4: Connect to Remote device using public address type and own address type is random.

  ConnectLE [Peer Public BD_ADDR] 1 1

For this example we will use option 2, The peripheral using Random address type and the central device using Public address type.

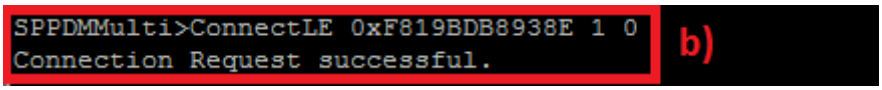

Figure 13-12. SPPDMMulti Demo Connect Successful

1. When a central successfully connects to the peripheral , both the central and peripheral will output LE_Connection_Complete and information about the current connection.

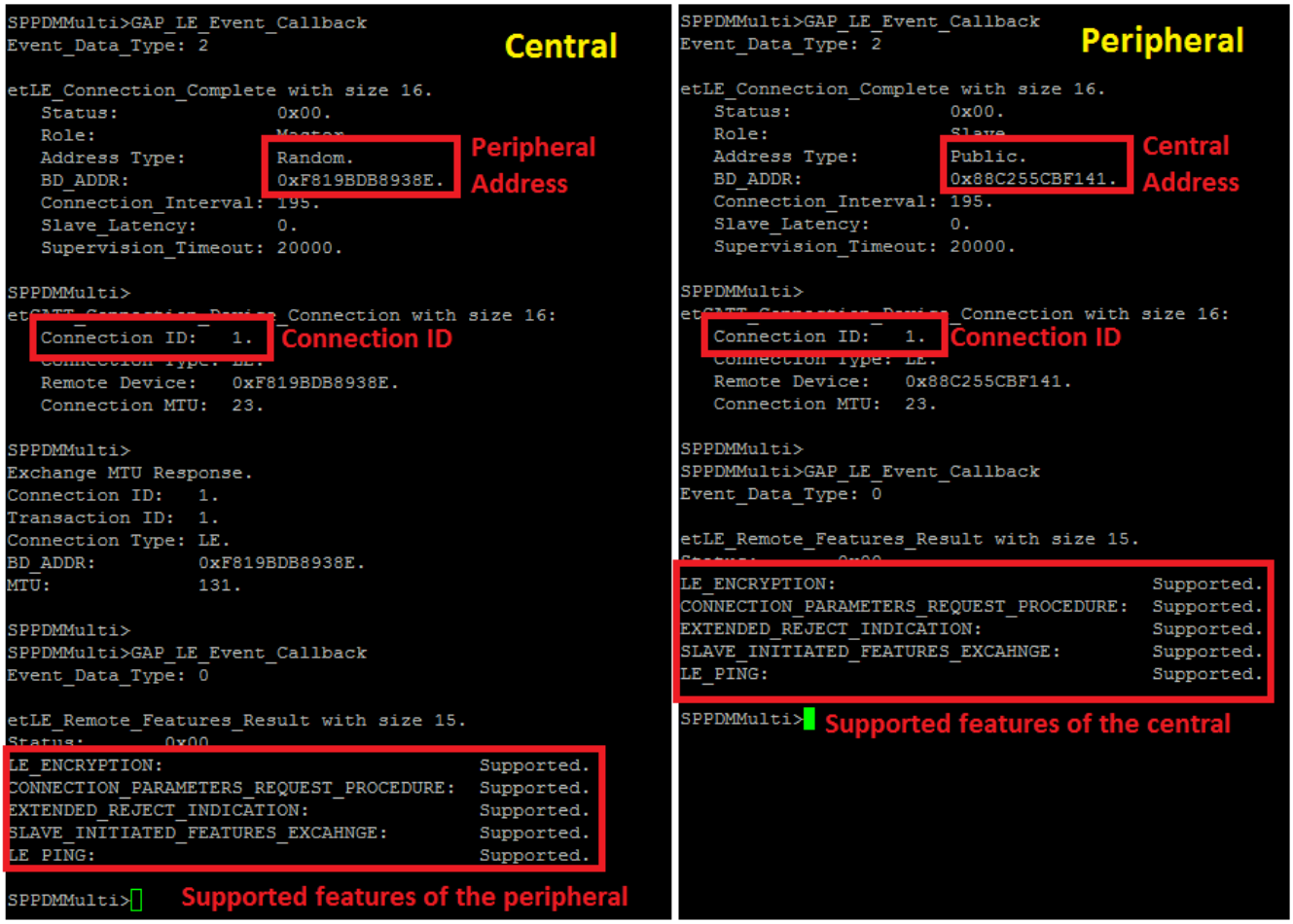

Figure 13-13. SPPDMMulti Demo Connection and Features

Note

The Supported features may change according to the Bluetooth device. For this example we used CC2564C for both sides.

Now we have a connection established and both devices are ready to send data to each other. Before the information can be sent we must first discover and configure the service in the Central terminal.

2. In order to discover the service handles we use the DiscoverLEService Command, type DiscoverLEService [BD_ADDR or Connection ID] [UUID Type] [0 when using BD_ADDR (optional) or 1 when using Connection ID (Mandatory)]. The handles that are discovered will be displayed in the terminal.

   The DiscoverLEServicecommand has multiple options to discover:

```
SPPDMMulti>DiscoverLEService
Usage: DiscoverLEService [BD_ADDR] [UUID_Type] [0 - Default] - Optional.
                         [Connection ID] [UUID_Type] [1]
When UUID_Type can be: 1 - SPPDMMulti, 2 - SPPLE,
                       16/32/128 - 16/32/128 bit UUIDs

Function Error.

SPPDMMulti>
```

Figure 13-14. SPPDMMulti Discover Services Command

- Option 1: Discover SPPDMMulti service which is using custom 32bit UUIDs a new feature from specification 4.1.

  This option will store the SPPDMMulti handles.

  DiscoverLEServicecommand [Peer BD_ADDR] [1]

  DiscoverLEServicecommand [Peer BD_ADDR] [1] [0]

  DiscoverLEServicecommand [Connection ID] [1] [1]
- Option 2: Discover SPPLE service which is using custom 128bit UUIDs.

  This option will store the SPPLE handles.

  DiscoverLEServicecommand [Peer BD_ADDR] [2]

  DiscoverLEServicecommand [Peer BD_ADDR] [2] [0]

  DiscoverLEServicecommand [Connection ID] [2] [1]
- Option 3 - Discover Generic services which are using 16/32/128bit UUIDs.

  This option will only print the handles.

  DiscoverLEServicecommand [Peer BD_ADDR] [16 or 32 or 128]

  DiscoverLEServicecommand [Peer BD_ADDR] [16 or 32 or 128] [0]

  DiscoverLEServicecommand [Peer BD_ADDR] [16 or 32 or 128] [1]

For this example we will use option 1, The peripheral is advertising with SPPDMMulti Handles so we need to store them and we will use the Connection ID method.

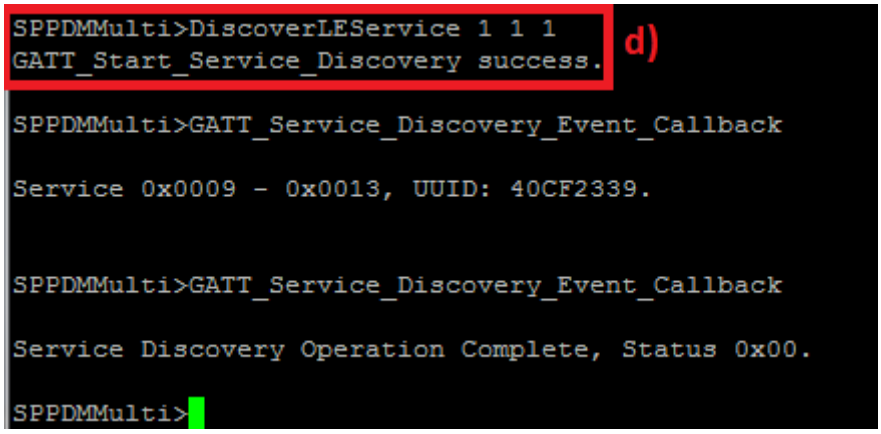

Figure 13-15. SPPDMMulti Demo Disocver LE Service Command

3. In order to configure the service we use the ConfigureLEService Command, type ConfigureLEService [BD_ADDR or Connection ID] [0 when using BD_ADDR (optional) or 1 when using Connection ID (Mandatory)].

```
SPPDMMulti>ConfigureLEService
Usage: ConfigureLEService [BD_ADDR] [0 - Default] - Optional.
                          [Connection ID] [1].

Function Error.

SPPDMMulti>
```

Figure 13-16. SPPDMMulti Demo Configure LE Service

```
SPPDMMulti>ConfigureLEService 1 1
SPPDMMULTI Service found on remote device, attempting to read Transmit Credits, and configured CCCDs.

SPPDMMulti>
Write Response.
Connection ID:   1.
Transaction ID:  12.
Connection Type: LE.
BD_ADDR:         0xF819BDB8938E.
Bytes Written:   2.

SPPDMMulti>
Write Response.
Connection ID:   1.
Transaction ID:  13.
Connection Type: LE.
BD_ADDR:         0xF819BDB8938E.
Bytes Written:   2.

SPPDMMulti>
```

e)

Figure 13-17. SPPDMMulti Demo Configure LE Results

4. After the LE connection is complete and configured we can use the ConnectedDevices command in order to receive a list of connected LE and/or BR/EDR devices (More information on this command can be found in paragraph 2.5.1.1b).

```
SPPDMMulti>ConnectedDevices
Using 1 LE Connections of maximum 4 available
Using 0 BR EDR Connections of maximum 1 available

Device Number 1:
    Remote BD_ADDR      : 0xF819BDB8938E
    LE Connected        : True
        Local LE Device Role    : Master
        Connection ID           : 1
    BR/EDR Connected    : False

Usage: ConnectedDevices [0 - Only Connected Devices (Default), 1 - All devices In the list] - optional.

SPPDMMulti>
```

One device connected via LE Connection

Figure 13-18. SPPDMMulti Demo Connected Devices

Note

The list above is from the Central side.

Connection over BR/EDR
Slave device initialization

We will setup the one of the boards (In our example, the one that we are running as a Peripheral device) as a slave so we need to open a SPP port on this device.

1. In order to open a SPP port, use OpenServer command with the port number that you want to open.

   For this example we will use port number 10.

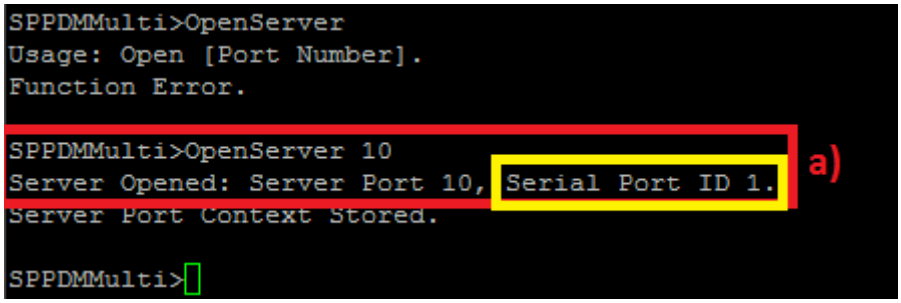

Figure 13-19. SPPDMMulti Demo Open Server Port

Note

Although the Server port number that was opened is 10, the Serial Port ID that will be used is 1. We allow only one BR/EDR connection so if you will open more than one ports, when the device will connect to a SPP port all the opened ports will close automatically

Master device connection creation

Now that the Slave device opened the SPP port, the Master can initiate the connection.

[Step 1 is optional if you already know the Bluetooth address of the device that you want to connect to]

1. The Master device can try to find which BR/EDR devices are in the vicinity using the command: Inquiry.

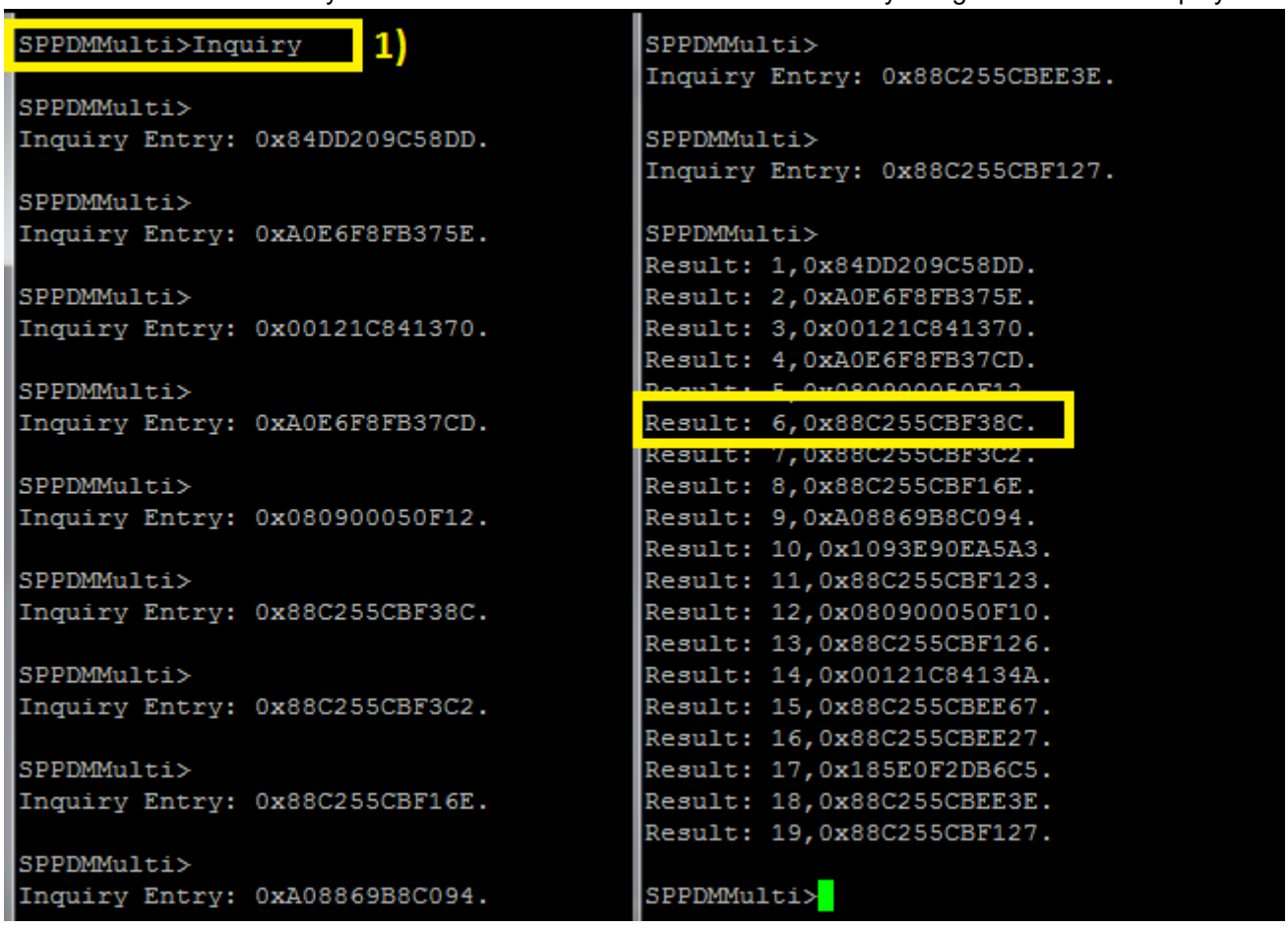

Figure 13-20. SPPDMMulti Demo Inquiry Results

2. a. Retrieve the Bluetooth address of the first board that was configured as a slave.
   b. In order to start the connection procedure type in the slave terminal OpenClient [Inquiry Index or BD_ADDR] [Server Port Number] [0 when using Inquiry Index (optional) or 1 when using BD_ADDR (Mandatory)].

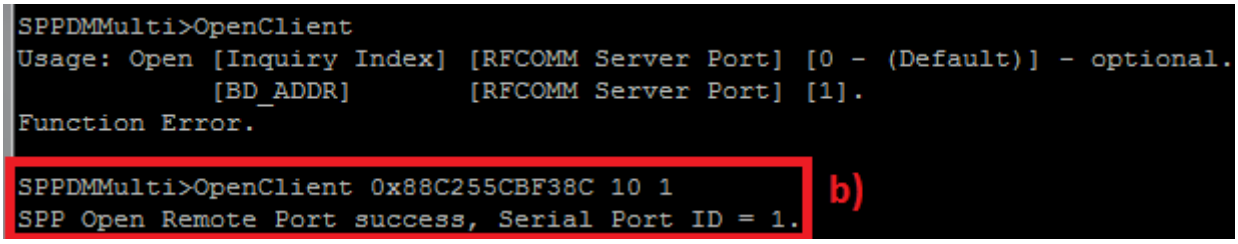

Figure 13-21. SPPDMMulti Demo Open Client Command

c. When a master successfully connects to the slave , both the master and slave will output the information about the current connection and will try to initiate role switch

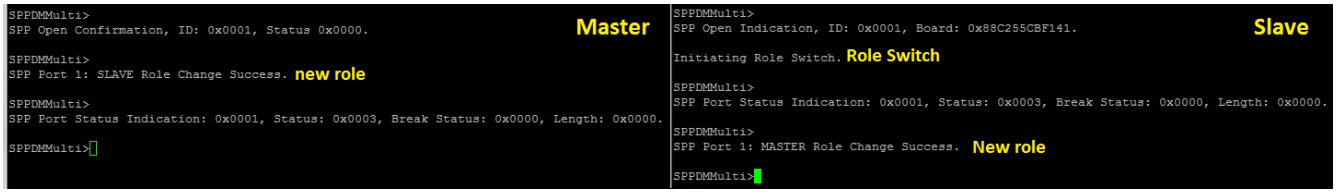

Figure 13-22. SPPDMMulti Demo Switching Roles

Now we have a connection established and both devices are ready to send data to each other.

d. After the BR/EDR connection is complete also we can use the ConnectedDevices command in order to receive a list of connected LE and/or BR/EDR devices (More information on this command can be found in paragraph 2.5.1.1b).

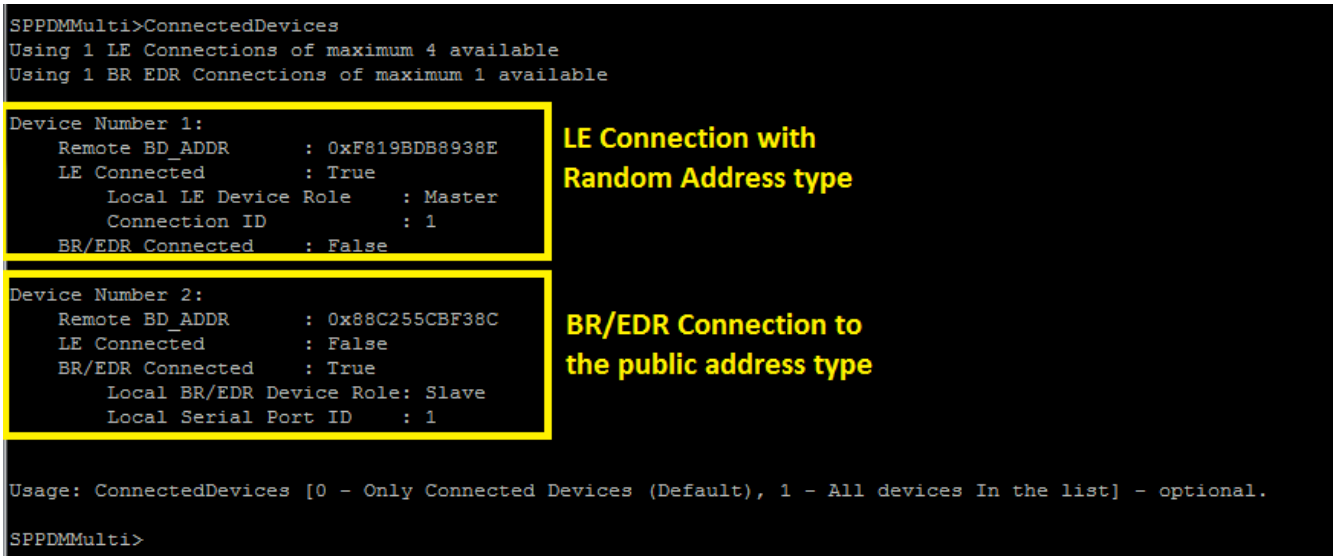

Figure 13-23. SPPDMMulti Demo Connected Devices Command 2

Note

The list above is from the LE Central terminal, the BR/EDR role is slave and not master because of the role switch. You can see 2 devices because the LE connection was to Random address type and the BR/EDR connections was to public address type if both connections were to the same address type you will see only 1 device.

Disconnect Connection

Disconnect over LE

1. In order to disconnect the connection, we use the DisconnectLE Command, type DisconnectLE [BD_ADDR or Connection ID] [0 when using BD_ADDR (optional) or 1 when using Connection ID (Mandatory)].

```
SPPDMMulti>DisconnectLE
Usage: DisconnectLE [BD_ADDR] [0 - Default] - Optional.
                    [Connection ID] [1]
Function Error.

SPPDMMulti>DisconnectLE 1 1
Disconnect Request successful.

SPPDMMulti>
etGATT_Connection_Device_Disconnection with size 12:
   Connection ID:   1.
   Connection Type: LE.
   Remote Device:   0x88C255CBF38C.

SPPDMMulti>GAP_LE_Event_Callback
Event_Data_Type: 3

etLE_Disconnection_Complete with size 9.
   Status: 0x00.
   Reason: 0x16.
   BD_ADDR: 0x88C255CBF38C.

SPPDMMulti>HCI Disconnection Complete Event, Status: 0x00, Connection Handle: 1025, Reason: 0x16

SPPDMMulti>
```

a)

Figure 13-24. SPPDMMulti Demo Disconnect BLE Command

Disconnect over BR/EDR

1. In order to disconnect the connection, in this case close the RFCOMM that was opened there are two options:

   Option 1, we use the CloseServer command from the slave , type CloseServer [Serial Port ID], if no Serial Port ID will be entered all opened ports will be closed.

   Option 2, we use the CloseClient command from the slave , type CloseClient Server [Serial Port ID], if no Serial Port ID will be entered all opened ports will be closed.

   For this example we will use option 2.

```
SPPDMMulti>CloseClient
Port Context Cleared.

SPPDMMulti>HCI Disconnection Complete Event, Status: 0x00, Connection Handle: 1, Reason: 0x13

SPPDMMulti>
```

a)

Figure 13-25. SPPDMMulti Demo Close Client Command

Pairing devices
LE Pairing

Note

The following steps describe how to configure the LE pairing operation, you can change any value that you want and jump to step c to start the LE pairing procedure with the changed values or the default values for the values that weren't changed and follow the instructions.

The default values are:

Pairability Mode - Pairable

I/O Capability - No Input/Output

Bonding Type - Bonding

MITM Requirement - Yes

SC Enable - Yes

P256 debug - No

1. In order to configure the Pairability mode, we need to use the SetPairabilityMode command and the mode that can be:

   0 = Non Pairable, 1 = Pairable

   For this example we will use Pairable mode (The same as in the default values).

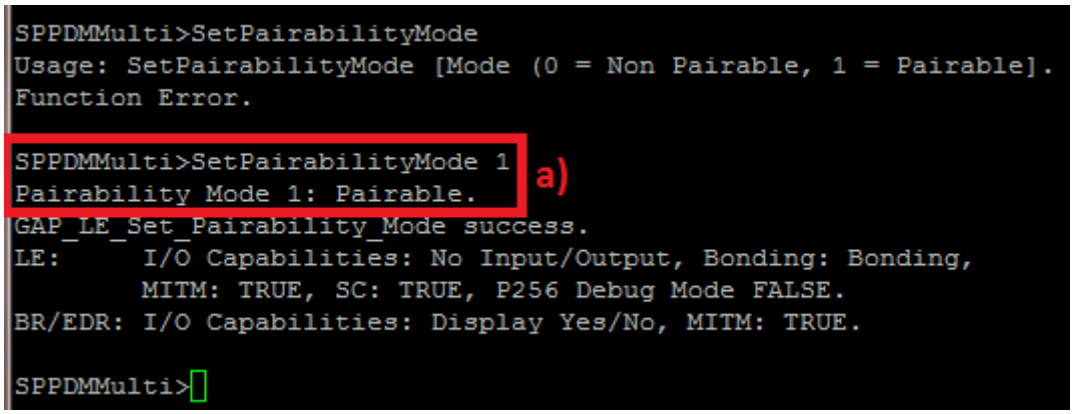

Figure 13-26. SPPDMMulti Demo Set Pair Mode

Note

After changing the pairability mode you will be able to see that default values.

2. In order to change the pairing parameters we can use the ChangePairingParameters Command, type ChangePairingParameters [I/O Capability] [Bonding Type] [MITM Requirement] [SC Enable] [P256 debug] where the options for this parameters are:

   I/O Capability (0 = Display Only, 1 = Display Yes/No, 2 = Keyboard Only, 3 = No Input/Output, 4 = Keyboard/Display)

   Bonding Type (0 = No Bonding, 1 = Bonding)

   MITM Requirement (0 = No, 1 = Yes)

   SC Enable (0 = No, 1 = Yes)

   P256 debug (0 = No, 1 = Yes)

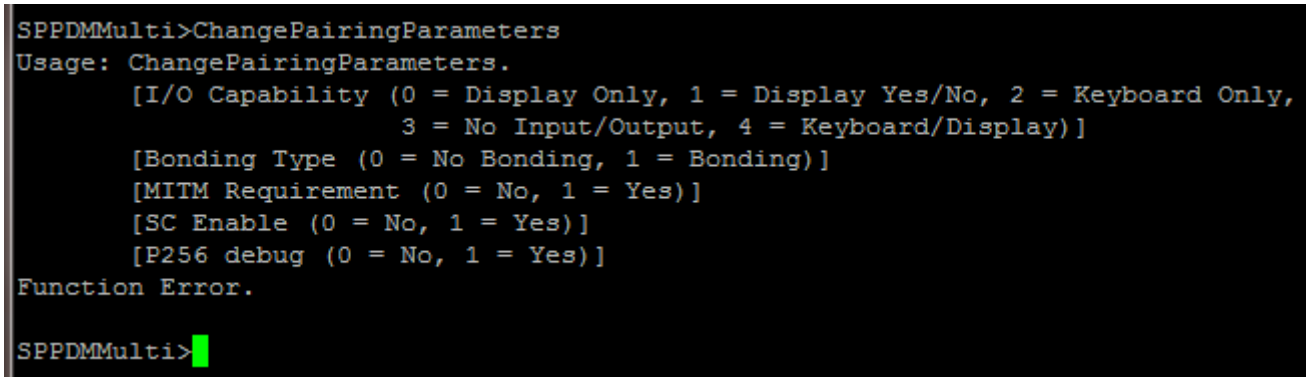

Figure 13-27. SPPDMMulti Demo Set Pairing Params

For this example we will use the default values.

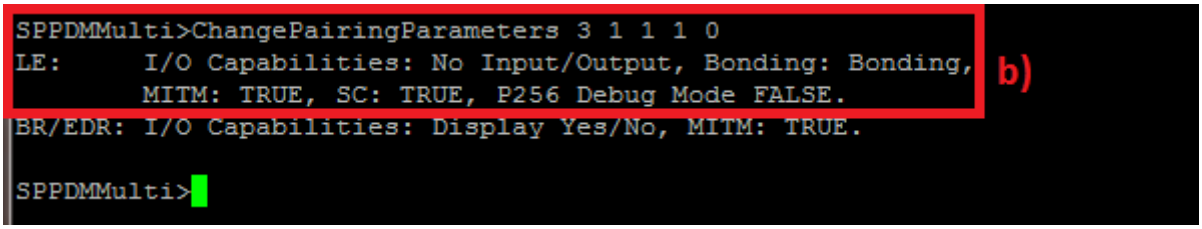

Figure 13-28. SPPDMMulti Demo Change Pairing Params

The LE Pairing procedure require an active LE connection, so after running the instructions in paragraph 2.1.1 you can start the LE pairing procedure.

3. In order to start the LE pairing procedure we need to use the PairLE command, so type PairLE [BD_ADDR or Connection ID] [0 when using BD_ADDR (optional) or 1 when using Connection ID (Mandatory)].

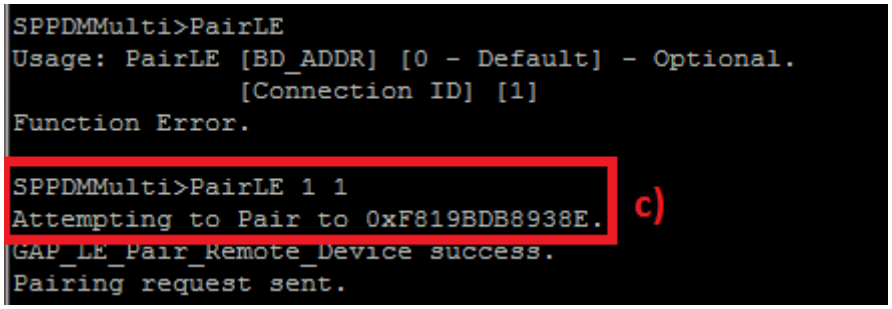

Figure 13-29. SPPDMMulti Demo Pair BLE Command

4. When a central successfully paired to the peripheral , both the central and peripheral will output LE_Authentication and information about the current connection.

Figure 13-30. SPPDMMulti Demo Secure Connection and Bonding

SPPDMMulti Demo Secure Connection and Bonding

Note

The following steps describe how to configure the BR/EDR pairing operation, you can change any value that you want and jump to steps c to start the BR/EDR pairing procedure with the changed values or the default values for the values that weren't changed and follow the instructions.

The default values are:

Pairability Mode - Pairable

I/O Capability - No Input/Output

MITM Requirement - Yes

1. In order to configure the Pairability mode, we need to use the SetCBPairabilityMode command and the mode that can be:

   0 = Non Pairable, 1 = Pairable, 2 = Pairable (Secure Simple Pairing

   For this example we will use Pairable mode (The same as in the default values).

```
SPPDMMulti>SetCBPairabilityMode
Usage: SetCBPairabilityMode [Mode (0 = Non Pairable, 1 = Pairable, 2 = Pairable (Secure Simple Pairing)].
Function Error.

SPPDMMulti>SetCBPairabilityMode 1
Pairability Mode Changed to pmPairableMode.

SPPDMMulti>
```

a)

Figure 13-31. SPPDMMulti Demo Set CB Pair Mode

2. In order to change the pairing parameters we can use the ChangeSimplePairingParameters Command, type ChangeSimplePairingParameters [I/O Capability] [MITM Requirement] where the options for this parameters are:

   I/O Capability (0 = Display Only, 1 = Display Yes/No, 2 = Keyboard Only, 3 = No Input/Output)

   MITM Requirement (0 = No, 1 = Yes)

```
SPPDMMulti>LESend
Usage: LESend [BD_ADDR] [Number of bytes to send] [0 - Default] - Optional.
              [Connection ID] [Number of bytes to send] [1]
Function Error.

SPPDMMulti>
```

Figure 13-32. SPPDMMulti Change Simple Pairing Params

For this example we will use the default values.

```
SPPDMMulti>ChangeSimplePairingParameters 3 1
LE:     I/O Capabilities: No Input/Output, Bonding: Bonding,
BR/EDR: I/O Capabilities: No Input/Output, MITM: TRUE.

SPPDMMulti>
```

b)

Figure 13-33. SPPDMMulti Demo Change Simple Pairing Params Command Response

The BR/EDR Pairing procedure require a non-active BR/EDR connection, so after running this instructions, run the instructions in paragraph 2.1.2.

3. In order to start the BR/EDR pairing procedure we need to use the Pair command, so type Pair [BD_ADDR or Connection ID] [Pairing type (0 = Dedicated - (Default), 1 = General)] [0 when using BD_ADDR (optional) or 1 when using Connection ID (Mandatory)].

```
SPPDMMulti>Pair
Usage: Pair [Inquiry Index] [0 = Dedicated - (Default), 1 = General] [0 - (Default)] - optional.
            [BD_ADDR]       [0 = Dedicated, 1 = General]             [1]
Function Error.

SPPDMMulti>Pair
```

Figure 13-34. SPPDMMulti Demo Pair Command

For this example we will use BD_ADDR and General pairing type.

```
SPPDMMulti>Pair 0x88C255CBF38C 1 1
GAP_Initiate_Bonding(General): Success.
```

c)

Figure 13-35. SPPDMMulti Demo Pair Command Response

4. When a central successfully paired to the peripheral , both the central and peripheral will output GAP_Authentication_Response and information about the current connection.

Note

if your devices weren't paired before, you may need to do additional steps - in this example we need to enter PINCodeResponse in both devices.

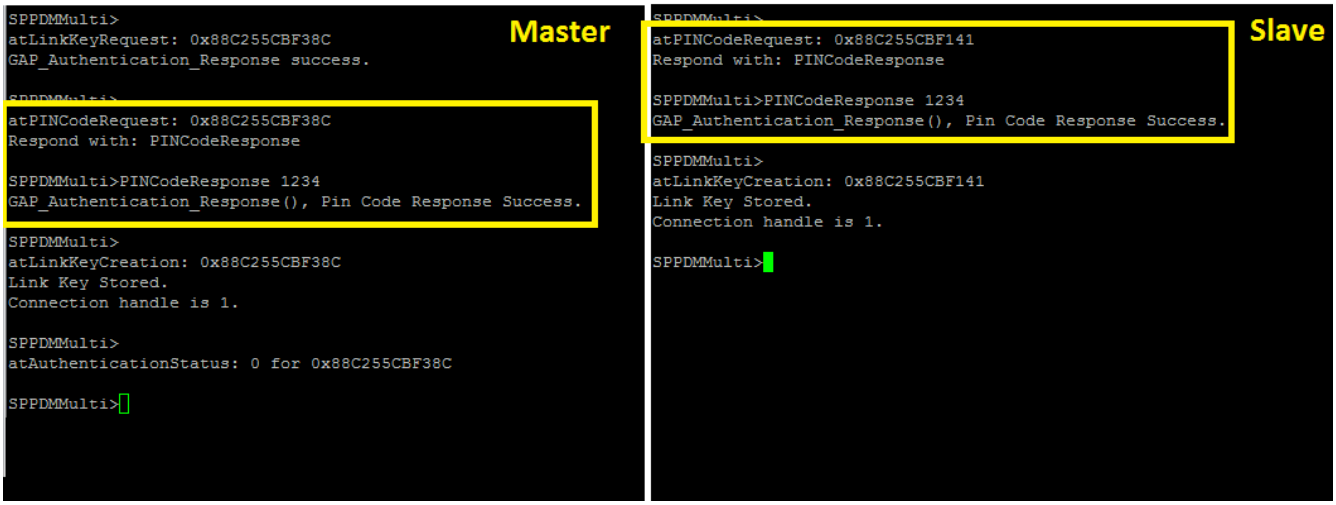

Figure 13-36. SPPDMMulti Demo PinCodeResponse

Un-Pairing devices

LE Un-Pairing

1. When there is no need any longer for the LTK stored in the device, you can delete it with the UnPairLE command with the following parameters [BD_ADDR or Connection ID] [0 when using BD_ADDR (optional) or 1 when using Connection ID (Mandatory)].

Note

Connection ID can be used only when the devices are connected, the LTK will be deleted but the connection will stay paired until disconnection procedure is requested.

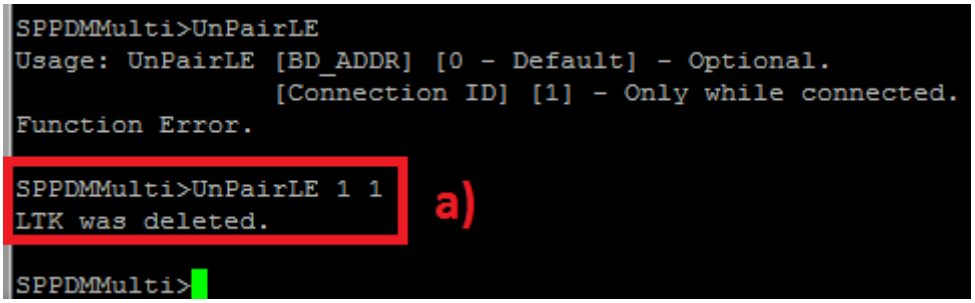

Figure 13-37. SPPDMMulti Demo BLE Unpair Command

BR/EDR Un-Pairing

1. When there is no need any longer for the Link Key stored in the device, you can delete it with the EndPairing command with the following parameters [Inquiry Index or BD_ADDR] [Server Port Number] [0 when using Inquiry Index (optional) or 1 when using BD_ADDR (Mandatory)].

Note

Using the EndPairing command will cancel the bonding between the devices and delete the Link Key, when using only Pair command with General method, the connection that was created will be disconnected, but if you had additional connection like RFCOMM port connected, the EndPairing command will not disconnect you.

```
SPPDMMulti>EndPairing
Usage: EndPairing [Inquiry Index] [0 - Option1].
                  [BD_ADDR]       [1]
Function Error.

SPPDMMulti>EndPairing 0x88C255CBF38C 1
GAP_End_Bonding success.

SPPDMMulti>HCI Disconnection Complete Event, Status: 0x00, Connection Handle: 1, Reason: 0x16

SPPDMMulti>
```

a)

Figure 13-38. SPPDMMulti Demo BT Unpair

Send and read Data

LE Data

1. In order to send data over LE, we need to use the LESend command with the following parameters [BD_ADDR or Connection ID] [Number of bytes to send] [0 when using BD_ADDR (optional) or 1 when using Connection ID (Mandatory)].

```
SPPDMMulti>LESend
Usage: LESend [BD_ADDR] [Number of bytes to send] [0 - Default] - Optional.
              [Connection ID] [Number of bytes to send] [1]
Function Error.

SPPDMMulti>
```

Figure 13-39. SPPDMMulti Demo BLE Send

For this example we will send 100 bytes and we will use the connection ID option.

```
SPPDMMulti>LESend 1 100 1

Send Complete, Sent 100.

SPPDMMulti>
```

a) Sender

```
SPPDMMulti>
Data Indication Event, Connection ID 1, Received 100 bytes.

SPPDMMulti>
```

Reciever

Figure 13-40. SPPDMMulti Demo BLE Send and Receive

2. In order to read the data that was sent, we need to use the LERead command with the following parameters [BD_ADDR or Connection ID] [0 when using BD_ADDR (optional) or 1 when using Connection ID (Mandatory)].

Note

The connection ID can be found easily in the Data Indication Event.

```
SPPDMMulti>LERead
Usage: LEREAD [BD_ADDR] [0 - Default] - Optional.
              [Connection ID] [1]

SPPDMMulti>LERead 1 1
Read: 100.
01234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789

SPPDMMulti>
```

b)

Figure 13-41. SPPDMMulti Demo BLE Read

BR/EDR Data

1. In order to send data over BR/EDR, there are two options to send the data.

   Option 1: Write command, This command will send only 10 bytes every use. To use this command type, Write [Serial Port ID].

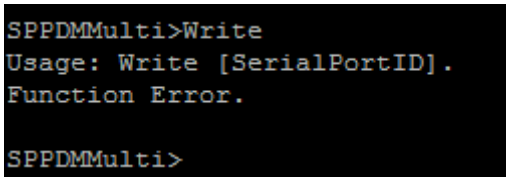

Figure 13-42. SPPDMMulti Demo BT Write Command

Option 2: CBSend command, This command will send the user input amount of bytes, This function can also disable or enable the sniff mode. To use this command type, CBSend [Number of Bytes to send] [Serial Port ID] [0 - Enable Automatic Sniff (Default) or 1 - Disable Automatic Sniff].

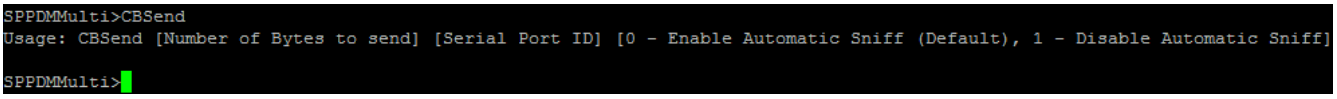

Figure 13-43. SPPDMMulti Demo CBSend Command

For this example we will use option 1 and send 10 bytes.

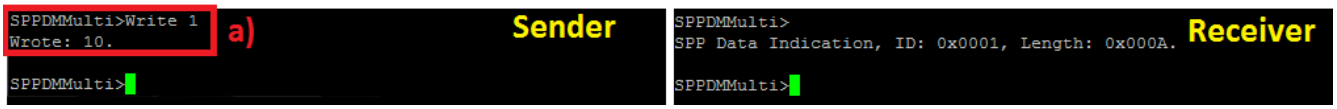

Figure 13-44. SPPDMMulti Demo BT Send and Receive

2. In order to read the data that was sent, we need to use the Read command with the following parameter [Serial Port ID].

Note

The Serial Port ID can be found easily in the SPP Data Indication event.

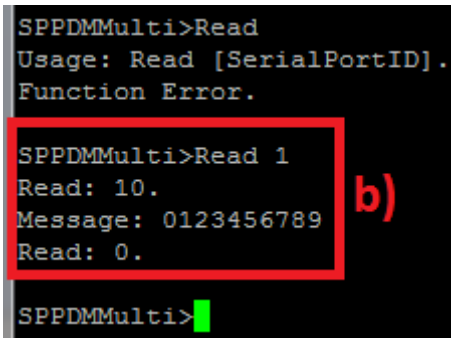

Figure 13-45. SPPDMMulti Demo BT Read

Additional Command

Note

The following paragraphs describe how to specific functions and are optional to use.

Generic features

LE Commands

Note

The following step describe how to change the connection parameters, you can change the values in the valid range or use the default parameters.

The default values are:

Connection_Interval_Min - 50ms (Range: 8..4000 in ms)

Connection_Interval_Max - 200ms (Range: 8..4000 in ms)

Minimum_Connection_Length - 0ms (Range: 0..65535 in ms)

Maximum_Connection_Length - 10000ms (Range: 0..65535 in ms)

Slave_Latency - 0ms (Range: 0..500 in ms)

Supervision_Timeout - 20000ms (Range: 100..32000 in ms)

ScanInterval - 100ms (Range: 3..10240 in ms)

ScanWindow - 100ms (Range: 3..10240 in ms)

1. In order to configure the connection parameters, we need to use the SetConnectionParameters command and fill the following parameters [Connection_Interval_Min] [Connection_Interval_Max] [Minimum_Connection_Length] [Maximum_Connection_Length] [Slave_Latency] (Range: 0..500 in ms) [Supervision_Timeout] [ScanInterval] [ScanWindow]

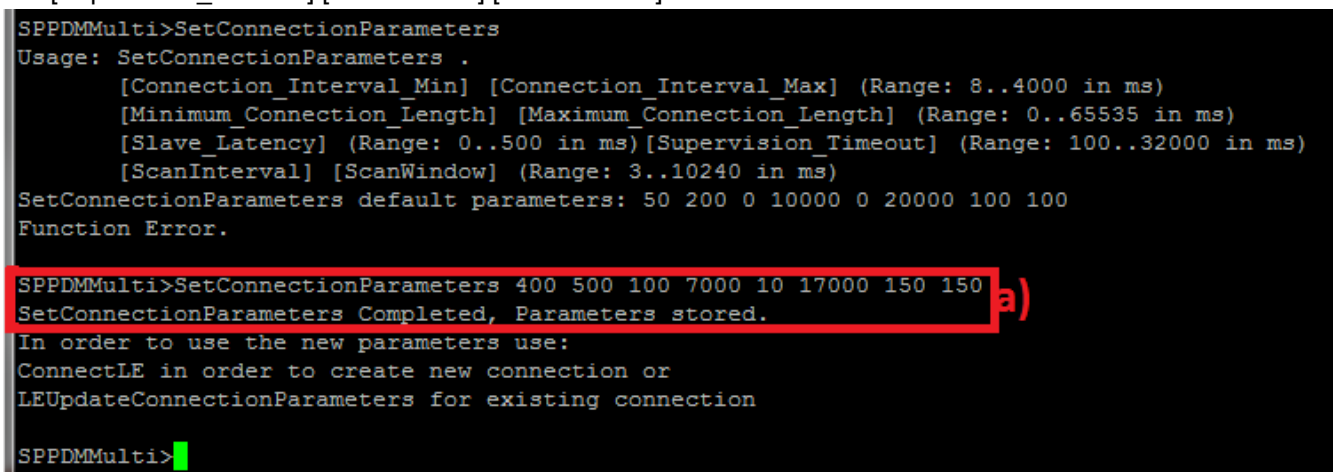

Figure 13-46. SPPDMMulti Demo Set Connection Parameters

Note

This command only stores the new connection parameters, in order to change the connection parameters you need to use additional command. ConnectLE when you want to create new connection (Paragraph 2.1.1) or LEUpdateConnectionParameters for existing connection (Paragraph 2.6.2a).

2. When the device is connected to multiple devices you can use ConnectedDevices in order to see the details of each connection.

   This command has two options to work.

   Option 1: List the contains only connected devices, For this option type ConnectedDevices with or without the parameter 0.

   In the Image below you can see that we are connected to three devices: To the first we are connected as a Master in LE (Central) and Slave in BR/EDR (Because of the role switch). To the second we are connected as a Master in LE (Central). To the third we are connected as a Slave in LE (Peripheral).

```
SPPDMMulti>ConnectedDevices
Using 3 LE Connections of maximum 4 available
Using 1 BR EDR Connections of maximum 1 available

Device Number 1:
    Remote BD_ADDR      : 0x88C255CBF38C
    LE Connected        : True
        Local LE Device Role    : Master
        Connection ID           : 1
    BR/EDR Connected    : True
        Local BR/EDR Device Role: Slave
        Local Serial Port ID    : 1

Device Number 2:
    Remote BD_ADDR      : 0x0017E9506518
    LE Connected        : True
        Local LE Device Role    : Master
        Connection ID           : 2
    BR/EDR Connected    : False

Device Number 3:
    Remote BD_ADDR      : 0x0017E951F154
    LE Connected        : True
        Local LE Device Role    : Slave
        Connection ID           : 3
    BR/EDR Connected    : False


Usage: ConnectedDevices [0 - Only Connected Devices (Default), 1 - All devices In the list] - optional.

SPPDMMulti>
```

Figure 13-47. SPPDMMulti Demo Connected Devices List

Option 2: List of all devices in the list, that means devices that was previously LE paired or opened SPP server ports.

In the Image below you can see the difference between option 1 and option 2. Option 1 shows that we are connected only to one device (Connection ID 2). Option 2 shows the full list. The green color shows devices that are not connected but we are paired with them (LTK is stored). The yellow color shows the device that is connected (Connection ID 2). The light blue color shows the opened SPP Server ports that are waiting to be connected.

Note

We allow only one BR/EDR connection so if you will open more than one ports, when the device will connect to a SPP port all the opened ports will close automatically.

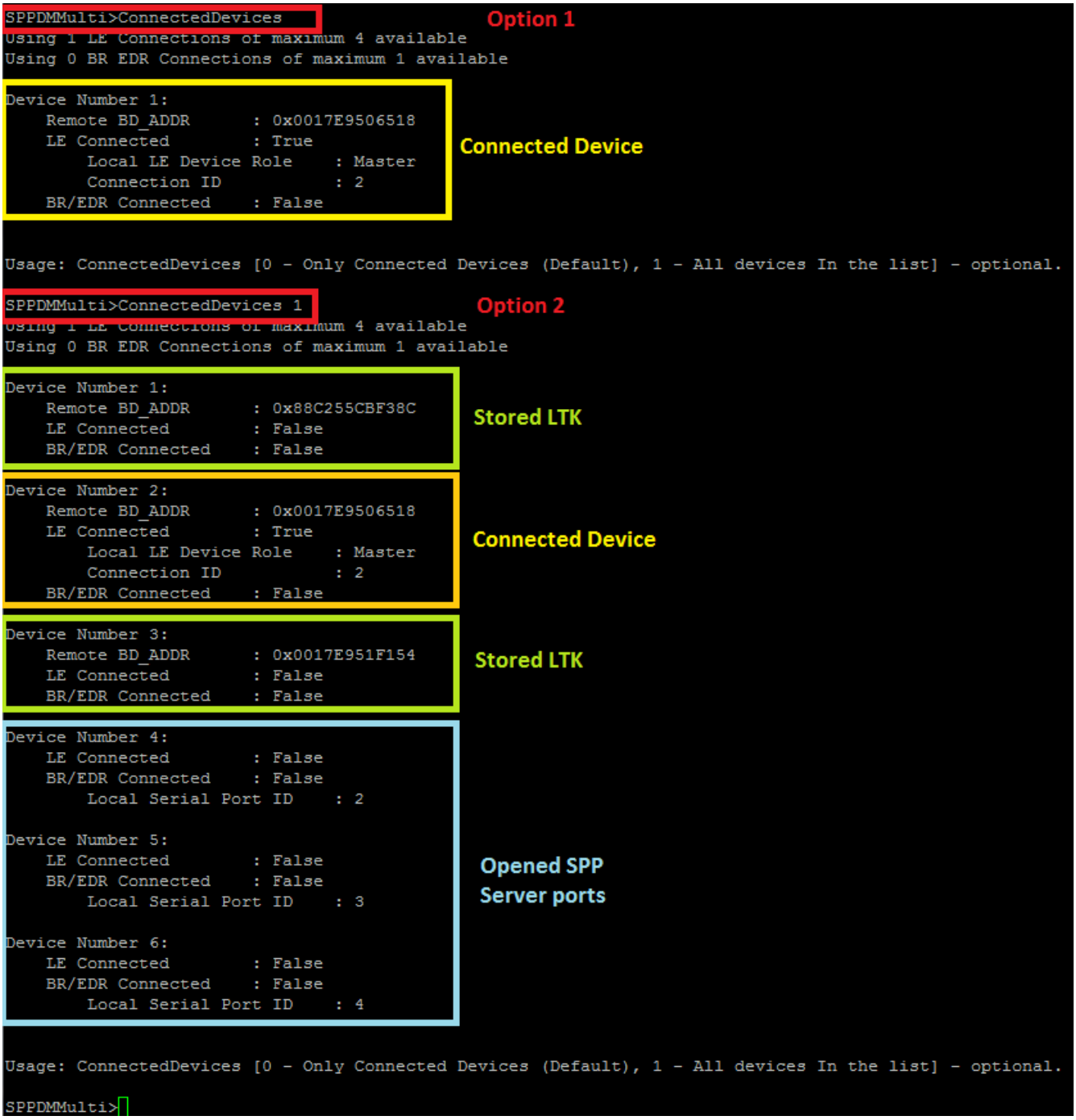

Figure 13-48. SPPDMMulti Demo Connected Devices Options

BR/EDR Commands

This sample application has Sniff mechanism that is used to free the baseband when BR/EDR connection is being used.

when using our sample and enabling this mode, when reading and writing over BR/EDR the device will exit and enter sniff automatically.

Note

The following step describe how to change the sniff parameters, you can change the values or use the default parameters.

The default values are:

MaxInterval - 600ms

MinInterval - 400ms

Attempts - 4

Timeout - 1

1. In order to configure the sniff parameters, we need to use the SetSniffParameters command and fill the following parameters [SerialPortID] [MaxInterval (ms)] [MinInterval (ms)] [Attempt] [Timeout].

Note

This command only stores the new sniff parameters for a specific Serial Port ID, that means that the device must be connected over BR/EDR (Paragraph 2.1.2).

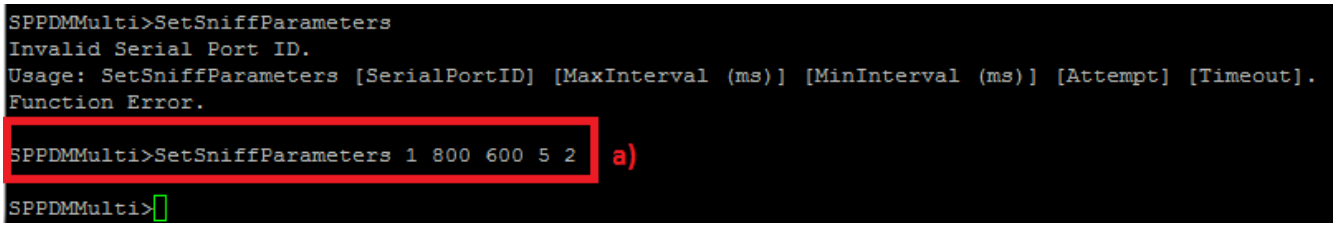

Figure 13-49. SPPDMMulti Demo Set Sniff Params

Note

This parameters that are stored, are stored only in the device that run this command and are not stored on the other device, you will need to run this command again on the other device as well or you can jump to step b if you want to initiate the sniff mode from the second device and enter the parameters directly.

2. In order to enter to sniff mode, we need to use the SniffMode command with the parameter [SerialPortID] all the following parameters are optional [MaxInterval (ms)] [MinInterval (ms)] [Attempt] [Timeout].

Note

If you set the optional parameters in step a they will be sent automatically when you will you Sniffmode [SerialPortID], if not and if you didn't filled them the default values will be sent.

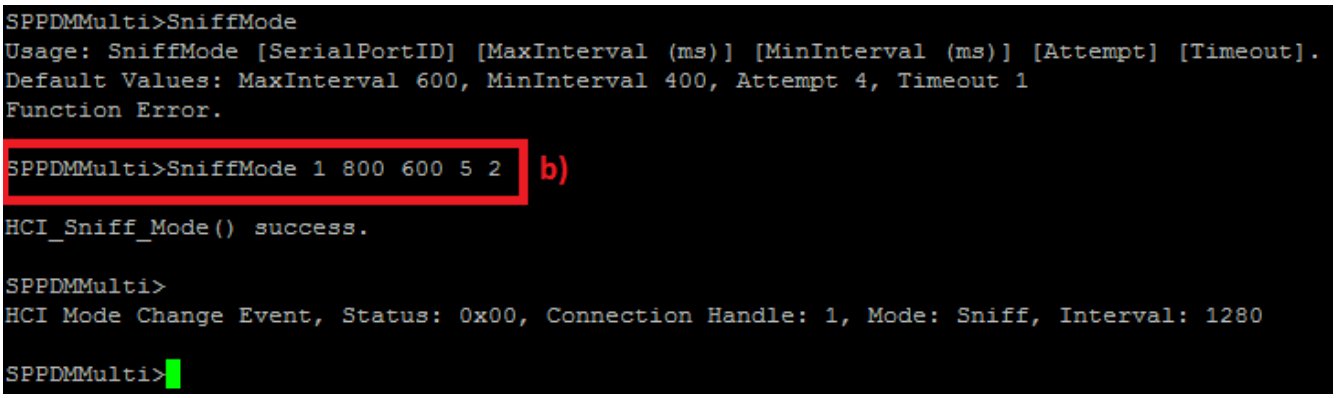

Figure 13-50. SPPDMMulti Demo Sniff Mode Command

Note

The Interval in HCI Mode Change Event is in baseband units and will displayed on both devices.

3. In order to exit the sniff mode, we need to use the ExitSniffMode command with the parameter [SerialPortID].

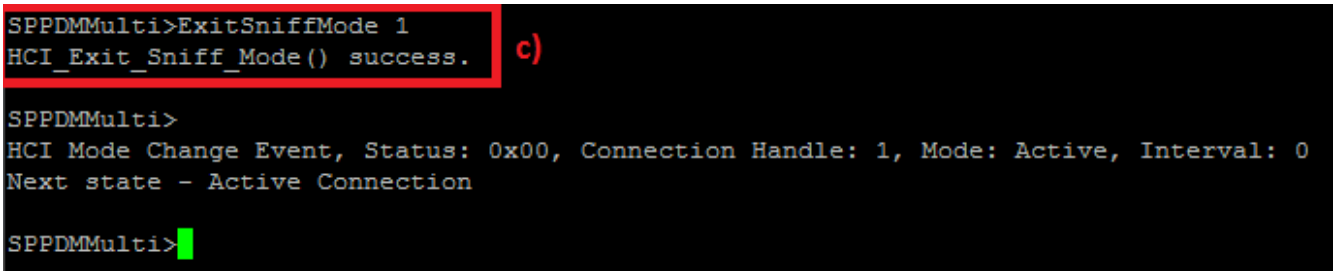

Figure 13-51. SPPDMMulti Demo Sniff Mode Option

Note

The HCI Mode Change Event will displayed on both devices.

New 4.1 and 4.2 Features

1. Until Bluetooth specification 4.1 only the central could send update connection parameters, in specification 4.1 the peripheral can request update connection parameters. While the connection is active, the central or the peripheral can ask to change the connection parameters after changing them in paragraph 2.6.1.1a. In order to send this request, use the LEUpdateConnectionParameters command with the following parameters [BD_ADDR or Connection ID] [0 when using BD_ADDR (optional) or 1 when using Connection ID (Mandatory)].

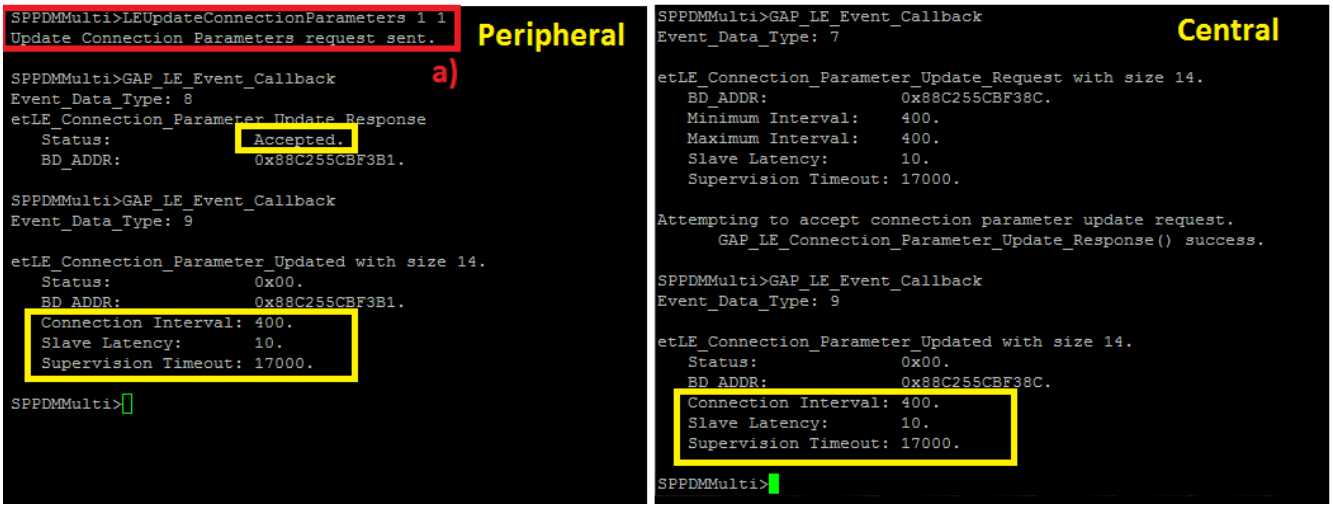

Figure 13-52. SPPDMMulti Demo Update Connection

Note

In the image you can see that the Central return status accepted to the peripheral and changed the parameters.

Both following command need the devices to be paired over LE, see paragraph 2.3.1 for instructions.

2. The SetAuthenticatedPayloadTimeout allows us to change the Authenticated Payload Timeout parameter from the default value that is 30 seconds to the user input value. In order to change it, type SetAuthenticatedPayloadTimeout with the following parameters [BD_ADDR or Connection ID] [Authenticated Payload Timeout] [0 when using BD_ADDR (optional) or 1 when using Connection ID (Mandatory)].

   The value Authenticated Payload Timeout is in ms units.

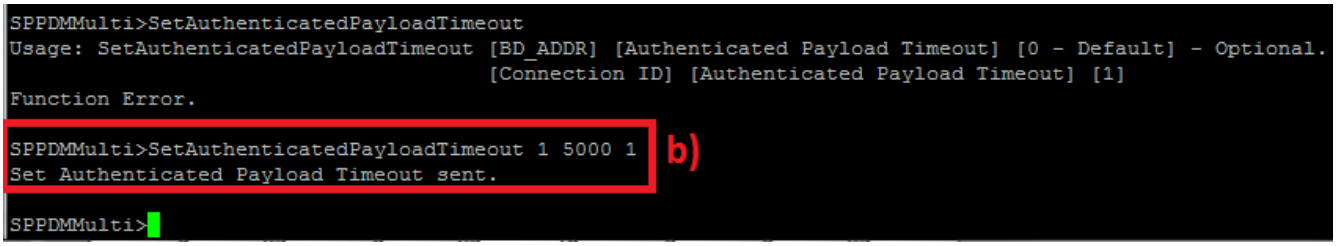

Figure 13-53. SPPDMMulti Demo Set Authentication Payload Timeout

3. The QueryAuthenticatedPayloadTimeout allows us to query the Authenticated Payload Timeout parameter of the connection. In order to query it, type QueryAuthenticatedPayloadTimeout with the following parameters [BD_ADDR or Connection ID] [0 when using BD_ADDR (optional) or 1 when using Connection ID (Mandatory)].

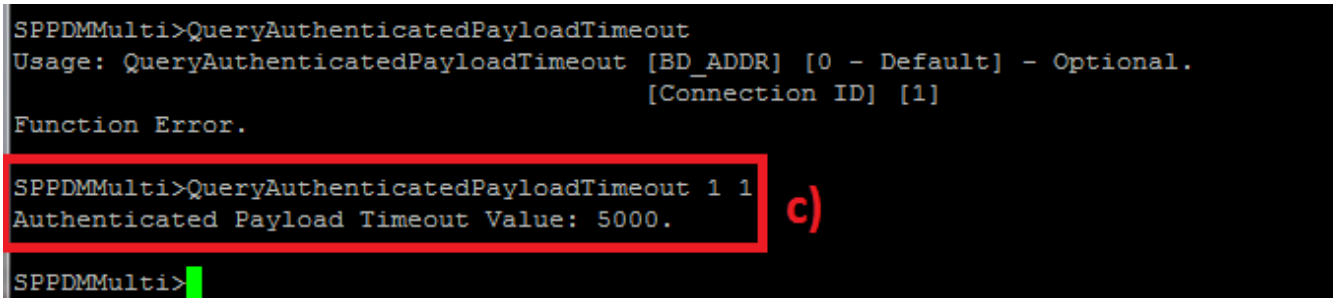

Figure 13-54. SPPDMMulti Demo Query Authencation Payload Timeout

13.3 Application Commands

ConnectedDevices

Description

The following function is responsible for displaying the information of all Current connected devices. This function will return zero on successful execution.

Parameters

This command requires can work in two options.

Option 1, zero parameters or the parameter 0, this option will display only the connected devices.

option 2, the parameter one, this option will display a list of all the stored devices.

Command Call Examples

“ConnectedDevices” Attempts to display only the connected devices.

“ConnectedDevices 0 ” Attempts to display only the connected devices.

“ConnectedDevices 1 ” Attempts to display a list of all the stored devices.

Possible Return Values

(0) Success.

RegisterSPPDMMULTI

Description

The following function is responsible for registering a SPPDMMULTI Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of registering a SPPDMMULTI Service.

Possible Return Values

(0) Successfully registered a SPPDMMULTI Service.

(-4) Function Error (on failure).

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE.

(-1000) BTGATT_ERROR_INVALID_PARAMETER.

(-1001) BTGATT_ERROR_NOT_INITIALIZED.

(-1004) BTGATT_ERROR_INVALID_BLUETOOTH_STACK_ID.

(-1005) BTGATT_ERROR_INSUFFICIENT_RESOURCES.

(-1011) BTGATT_ERROR_INVALID_SERVICE_TABLE_FORMAT.

(-1013) BTGATT_ERROR_INSUFFICIENT_HANDLES.

API Call

GATT_Register_Service(BluetoothStackID, SPPDMMULTI_SERVICE_FLAGS, SPPDMMULTI_SERVICE_ATTRIBUTE_COUNT, (GATT_Service_Attribute_Entry_t *)SPPDMMULTI_Service, &ServiceHandleGroup, GATT_ServerEventCallback, 0)

API Prototype

int BTPSAPI GATT_Register_Service(unsigned int BluetoothStackID, Byte_t ServiceFlags, unsigned int NumberOfServiceAttributeEntries, GATT_Service_Attribute_Entry_t *ServiceTable, GATT_Attribute_Handle_Group_t *ServiceHandleGroupResult, GATT_Server_Event_Callback_t ServerEventCallback, unsigned long CallbackParameter)

Description of API

The following function is provided to allow a means to add a GATT Service to the local GATT Database. The first parameter is Bluetooth stack ID of the Bluetooth Device. The second parameter is a bit mask field that specifies the type of service being registered, which must be non-zero (i.e. at least one bit must be set). The third parameter is the number of entries in the service attribute array that is pointed to by the fourth parameter. The fourth parameter is an array that contains the attributes for the service being registered. The next parameter is a pointer to a buffer that will store the attribute handle range of the registered service. The final two parameters specify the GATT server callback and callback parameter that will be used whenever a client request to the GATT server cannot be satisified internally by the local GATT module. This function will return a positive non-zero service ID if successful, or a negative return error code if there was an error. If this function returns success then the ServiceHandleGroupResult buffer will contain the service's attribute handle range.

DiscoverLEService

Description

The following function is responsible for performing the Service Discovery Operation. The User can Discover the following options:

SPPDMMULTI, SPPLE or discover all the 16/32/128 bit UUID's from The Peer device. This function will return zero on successful execution and a negative value on errors.

Parameters

This command requires at least two parameters.

At least two parameter when using as the first parameter the Bluetooth Address of the remote device that is connected, the second parameters is the UUID type.

Or three parameters when the first parameter is the connection ID of the connection with the remote device, the second parameters is the UUID type,the third parameter is 1 and it is mandatory.

The UUID type can be: 1 - SPPDMMulti, 2 - SPPLE, 16/32/128 - 16/32/128 bit UUIDs

Command Call Examples

DiscoverLEService 001bdc05b617 0” Attempts to discover SPPDMMULTI services of the Bluetooth Device with the BD_ADDR of 001bdc05b617.

DiscoverLEService 000275e126FF 1 0” Attempts to discover SPPLE services of the Bluetooth Device with the BD_ADDR of 000275e126FF.

DiscoverLEService 2 0 1” Attempts to discover SPPDMMULTI services of the connection with the connection id of 2.

DiscoverLEService 1 32 1” Attempts to discover 32bit UUID's services of the connection with the connection id of 1.

Possible Return Values

(0) Successfully started a SPP LE Service Discovery.

(-4) Function Error (on failure).

(-6) INVALID_PARAMETERS_ERROR.

(-8) INVALID_STACK_ID_ERROR.

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE.

(-1000) BTGATT_ERROR_INVALID_PARAMETER.

(-1005) BTGATT_ERROR_INSUFFICIENT_RESOURCES.

(-1010) BTGATT_ERROR_OUTSTANDING_REQUEST_EXISTS.

API Call

GATT_Start_Service_Discovery(BluetoothStackID, DeviceInfo->LEContextInfo.ConnectionID, (General_Discovery)?0:(sizeof(UUID)/sizeof(GATT_UUID_t)), (General_Discovery)?NULL:UUID, GATT_Service_Discovery_Event_Callback, DiscoveryType)

API Prototype

int BTPSAPI GATT_Start_Service_Discovery(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GATT_Service_Discovery_Event_Callback_t ServiceDiscoveryCallback, unsigned long CallbackParameter)

Description of API

The following function is used to initiate a GATT Service Discovery process to the specified GATT connection. The function takes as its first parameter the BluetoothStackID that is associated with the Bluetooth Device. The second parameter is the Connection ID of the remote device that is to be searched. The third and fourth parameters specify an optional list of UUIDs to search for. The final two parameters define the Callback function and parameter to use when the service discovery is complete. The function returns zero on success and a negative return value if there was an error.

Note

The NumberOfUUID and UUIDList are optional. If they are specified then they specify the list of services that should be searched for. Only services with UUIDs that are in this list will be discovered.

Note

Only 1 service discovery operation can be outstanding at a time.

ConfigureLEService

Description

The following function is responsible to configure a SPPDMMULTI Service on a remote device. This function will return zero on successful execution and a negative value on errors. The following function enables notifications of the proper characteristics based on a specified handle; depending what the device role for SPPDMMULTI is, server or client, the API function that is called is either a GATT_Handle_Value_Notification or a

GATT_Write_Without_Response_Request; which notifies the receiving credit characteristic or sends a write with out response packet to the transmission credit characteristic respectively.

Parameters

This command requires at least one parameter.

At least one parameter when using as the first parameter the Bluetooth Address of the remote device that is connected, the second parameters in this option is 0 and optional.

Or two parameters when the first parameter is the connection ID of the connection with the remote device, the second parameter is 1 and it is mandatory.

Command Call Examples

“ConfigureSPPDMMULTI 001bdc05b617” Attempts to configure services of the Bluetooth Device with the BD_ADDR of 001bdc05b617.

“ConfigureSPPDMMULTI 000275e126FF 0” Attempts to configure services of the Bluetooth Device with the BD_ADDR of 000275e126FF.

“ConfigureSPPDMMULTI 1 1” Attempts to configure services of the connection with the connection id of 1.

Possible Return Values

(0) Successfully configured a SPPDMMULTI Service.

(-4) Function Error (on failure).

(-6) INVALID_PARAMETERS_ERROR.

(-8) INVALID_STACK_ID_ERROR.

(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID.

(-26) BTPS_ERROR_L2CAP_NOT_INITIALIZED.

(-37) BTPS_ERROR_CHANNEL_NOT_IN_OPEN_STATE.

(-38) BTPS_ERROR_INVALID_CID.

(-41) BTPS_ERROR_NEGOTIATED_MTU_EXCEEDED.

(-47) BTPS_ERROR_INVALID_CID_TYPE.

(-65) BTPS_ERROR_INSUFFICIENT_BUFFER_SPACE.

(-98) BTPS_ERROR_DEVICE_NOT_CONNECTED.

(-1000) BTGATT_ERROR_INVALID_PARAMETER.

(-1001) BTGATT_ERROR_NOT_INITIALIZED.

(-1004) BTGATT_ERROR_INVALID_BLUETOOTH_STACK_ID.

(-1005) BTGATT_ERROR_INSUFFICIENT_RESOURCES.

(-1009) BTGATT_ERROR_INVALID_CONNECTION_ID.

API Call

GATT_Write_Request(BluetoothStackID, ConnectionID, ClientConfigurationHandle, sizeof(Buffer), &Buffer, ClientEventCallback, 0)

GATT_Handle_Value_Notification(BluetoothStackID, SPPDMMULTIServiceID, ConnectionID, SPPDMMULTI_RX_CREDITS_CHARACTERISTIC_ATTRIBUTE_OFFSET, WORD_SIZE, (Byte_t *)&Credits)

GATT_Write_Without_Response_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.Tx_Credit_Characteristic, WORD_SIZE, &Credits)

API Prototype

int BTPSAPI GATT_Write_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue, GATT_Client_Event_Callback_t ClientEventCallback, unsigned long CallbackParameter)

int BTPSAPI GATT_Handle_Value_Notification(unsigned int BluetoothStackID, unsigned int ServiceID, unsigned int ConnectionID, Word_t AttributeOffset, Word_t AttributeValueLength, Byte_t *AttributeValue)

int BTPSAPI GATT_Write_Without_Response_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue)

Description of API

The first of these API functions is provided to allow a means of performing a write request to a remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write the value of, followed by the length of the value (in bytes), followed by the the actual value data to write. The final two parameters specify the GATT client event callback function and callback parameter (respectively) that will be called when a response is received from the remote device. This function will return the positive, non-zero, Transaction ID of the request or a negative error code.

The second of these API functions allows a means of sending a Handle/Value notification to a remote GATT client. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack. The second parameter is the service ID of the service that is sending the Handle/Value notification. The third parameter specifies the connection ID of the connection to send the Handle/Value notification to. The fourth parameter specifies the offset in the service table (registered via the call to the GATT_Register_Service() function) of the attribute that is being notified. The fifth parameter is the length (in bytes) of the attribute value that is being notified. The sixth parameter is a pointer to the actual attribute value to notify. This function will return a non-negative value that represents the actual length of the attribute value that was notified, or a negative return error code if there was an error.

The third of these API functions is provided to allow a means of performing a write without response request to remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write, followed by the length of the value data to write (in bytes), followed by the actual value to write. This function will return the number of bytes written on success or a negative error code.

LESend

Description

The following function is responsible for sending a number of characters to a remote device to which a connection exists. The function receives a parameter that indicates the number of byte to be transferred. This function will return zero on successful execution and a negative value on errors. Depending what the device role for SPPDMMULTI is, server or client, the API function that is called is either a GATT_Handle_Value_Notification or a GATT_Write_Without_Response_Request; which notifies the receiving credit characteristic or sends a write with out response packet to the transmission credit characteristic respectively.

Parameters

This command requires at least two parameters.

At least two parameters when using as the first is the remote Bluetooth address of the device you are sending to. The second is the number of bytes to send. This value has to be greater than 10. The third parameter in this option is 0 and optional.

Or three parameters when the first parameter is the connection ID of the connection with the remote device you are sending to. The second is the number of bytes to send. This value has to be greater than 10. The third parameter is 1 and it is mandatory.

Command Call Examples

"LeSend 0017E7FEFD7C 100" Attempts to send 100 bytes of data to 0017E7FEFD7C.

"LeSend B8FFFEAF1CAD 25 0" Attempts to send 25 bytes of data to B8FFFEAF1CAD.

"LeSend 1 25 1" Attempts to send 25 bytes of data to the remote device with that is connected on connection ID 1.

Possible Return Values

(0) Successfully Sent Data.

(-4) FUNCTION_ERROR.

(-6) INVALID_PARAMETERS_ERROR.

(-8) INVALID_STACK_ID_ERROR.

API Call

GATT_Handle_Value_Notification(BluetoothStackID, SPPDMMULTIServiceID, ConnectionID, SPPDMMULTI_TX_CHARACTERISTIC_ATTRIBUTE_OFFSET, (Word_t)DataCount, SPPDMMULTIBuffer)

GATT_Write_Without_Response_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.Rx_Characteristic, (Word_t)DataCount, SPPDMMULTIBuffer)

API Prototype

int BTPSAPI GATT_Handle_Value_Notification(unsigned int BluetoothStackID, unsigned int ServiceID, unsigned int ConnectionID, Word_t AttributeOffset, Word_t AttributeValueLength, Byte_t *AttributeValue)

int BTPSAPI GATT_Write_Without_Response_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue)

Description of API

The first of these API functions allows a means of sending a Handle/Value notification to a remote GATT client. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack. The second parameter is the service ID of the service that is sending the Handle/Value notification. The third parameter specifies the connection ID of the connection to send the Handle/Value notification to. The fourth parameter specifies the offset in the service table (registered via the call to the GATT_Register_Service() function) of the attribute that is being notified. The fifth parameter is the length (in bytes) of the attribute value that is being notified. The sixth parameter is a pointer to the actual attribute value to notify. This function will return a non-negative value that represents the actual length of the attribute value that was notified, or a negative return error code if there was an error.

The second of these API functions is provided to allow a means of performing a write without response request to remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write, followed by the length of the value data to write (in bytes), followed by the actual value to write. This function will return the number of bytes written on success or a negative error code.

LERead

Description

The following function is responsible for reading data sent by a remote device to which a connection exists. This function will return zero on successful execution and a negative value on errors. Depending what the device role for SPPDMMULTI is, server or client, the API function that is called is either a GATT_Handle_Value_Notification or a GATT_Write_Without_Response_Request; which notifies the receiving credit characteristic or sends a write with out response packet to the transmission credit characteristic respectively.

Parameters

This command requires at least one parameter.

At least one parameters when using as the first is the remote Bluetooth address of the device you are reading from. The second parameter in this option is 0 and optional.

Or two parameters when the first parameter is the connection ID of the connection with the remote device you are reading from. The second parameter is 1 and it is mandatory.

Command Call Examples

“LeRead 001bdc05b617” Attempts to read data of the Bluetooth Device with the BD_ADDR of 001bdc05b617.

“LeRead 000275e126FF 0” Attempts to read data of the Bluetooth Device with the BD_ADDR of 000275e126FF.

“LeRead 1 1” Attempts to read data of the remote device is connected on connection ID 1.

Possible Return Values

(0) Successfully Read Data.

(-6) INVALID_PARAMETERS_ERROR.

API Call

GATT_Handle_Value_Notification(BluetoothStackID, SPPDMMULTIServiceID, ConnectionID, SPPDMMULTI_RX_CREDITS_CHARACTERISTIC_ATTRIBUTE_OFFSET, WORD_SIZE, (Byte_t *)&Credits)

GATT_Write_Without_Response_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.Tx_Credit_Characteristic, WORD_SIZE, &Credits)

API Prototype

int BTPSAPI GATT_Handle_Value_Notification(unsigned int BluetoothStackID, unsigned int ServiceID, unsigned int ConnectionID, Word_t AttributeOffset, Word_t AttributeValueLength, Byte_t *AttributeValue)

int BTPSAPI GATT_Write_Without_Response_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue)

Description of API

The first of these API functions allows a means of sending a Handle/Value notification to a remote GATT client. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack. The second parameter is the service ID of the service that is sending the Handle/Value notification. The third parameter specifies the connection ID of the connection to send the Handle/Value notification to. The fourth parameter specifies the offset in the service table (registered via the call to the GATT_Register_Service() function) of the attribute that is being notified. The fifth parameter is the length (in bytes) of the attribute value that is being notified. The sixth parameter is a pointer to the actual attribute value to notify. This function will return a non-negative value that represents the actual length of the attribute value that was notified, or a negative return error code if there was an error.

The second of these API functions is provided to allow a means of performing a write without response request to remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write, followed by the length of the value data to write (in bytes), followed by the actual value to write. This function will return the number of bytes written on success or a negative error code.

Loopback

Description

The Loopback command is responsible for setting the application state to support loopback mode. This command will return zero on successful execution and a negative value on errors.

Parameters

This command requires one parameter which indicates if loopback should be supported. 0 = loopback not active, 1 = loopback active.

Command Call Examples

"Loopback 0" sets loopback support to inactive.

"loopback 1" sets loopback support to active.

Possible Return Values

(0) Successfully set loopback support.

(-6) INVALID_PARAMETERS_ERROR.

DisplayRawModeData

Description

The following function is responsible for setting the application state to support displaying Raw Data. This function will return zero on successful execution and a negative value on errors.

Parameters

This command accepts one parameter which indicates if displaying raw data mode should be supported. 0 = Display Raw Data Mode inactive, 1 = Display Raw Data active.

Command Call Examples

"DisplayRawModeData 0" sets Display Raw Mode support inactive.

"DisplayRawModeData 1" sets Display Raw Mode support active.

Possible Return Values

(0) Successfully sets Display Raw Data Mode support.

(-6) INVALID_PARAMETERS_ERROR.

AutomaticReadMode

Description

The AutomaticReadMode command is responsible for setting the application state to support Automatically reading all data that is received through SPP. This function will return zero on successful execution and a negative value on errors.

Parameters

This command accepts one parameter which indicates if automatic read mode should be supported. 0 = Automatic Read Mode inactive, 1 = Automatic Read Mode active.

Command Call Examples

"AutomaticReadMode 0" sets Automatic Read Mode support to inactive.

"AutomaticReadMode 1" sets Automatic Read Mode support to active.

Possible Return Values

(0) Successfully set Automatic Read Mode support.

(-6) INVALID_PARAMETERS_ERROR.

14 ANP Demo Guide

14.1 Demo Overview

Note

The same instructions can be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms.

The Alert Notification profile allows a Client device to get alerts of different types (like Email, SMS, Voicemail) and also have separate counts of new and unread Alerts. There are two roles defined in this profile. The first is the the Server, which is typically a Bluetooth enabled phone that transmits these alerts. The second is the Client which can be a device like a smart watch, an auxillary display or even a tablet that gets an alert everytime an event happens on the Server.

This application allows the user to use a console to use Bluetooth Low Energy (BLE) to establish connection between two BLE devices, notify new and unread alerts and change the supported alert categories for new and unread alerts.

It is recommended that the user visits the kit setup Getting Started Guide for MSP430, Getting Started Guide for Tiva, Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as **MSP-EXP430F5438 USB - Serial Port (COM x)**, **Tiva Virtual COM Port (COM x)**, **XDS110 Class Application/User UART (COM x)** for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud (9600 for MSP430), 8, n, 1. Once connected, reset the device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands. This device will become the Server.

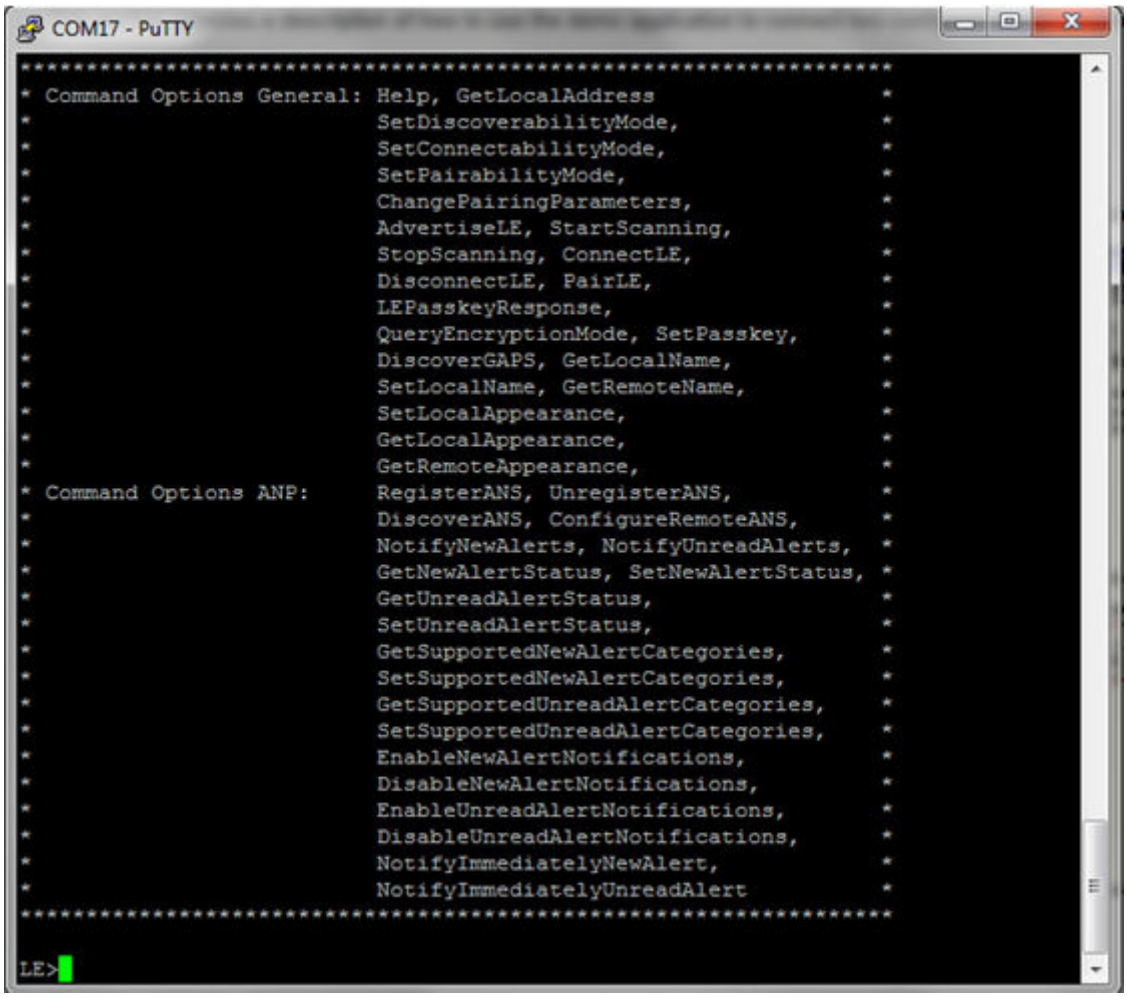

Figure 14-1. ANP Demo ANS Terminal

14.2 Demo Application

The demo application provides a description on how to use the demo application to connect two configured boards and communicate over BluetoothLE. The included application registers a custom service on a board when the stack is initialized.

Device 1 (Server) setup on the demo application

1. To start, one of the devices has to have the Alert Notification Service running on it. It can be started by running **Register ANS**.
2. Next, the device acting as a Server needs to advertise to other devices. This can be done by running **AdvertiseLE 1**.

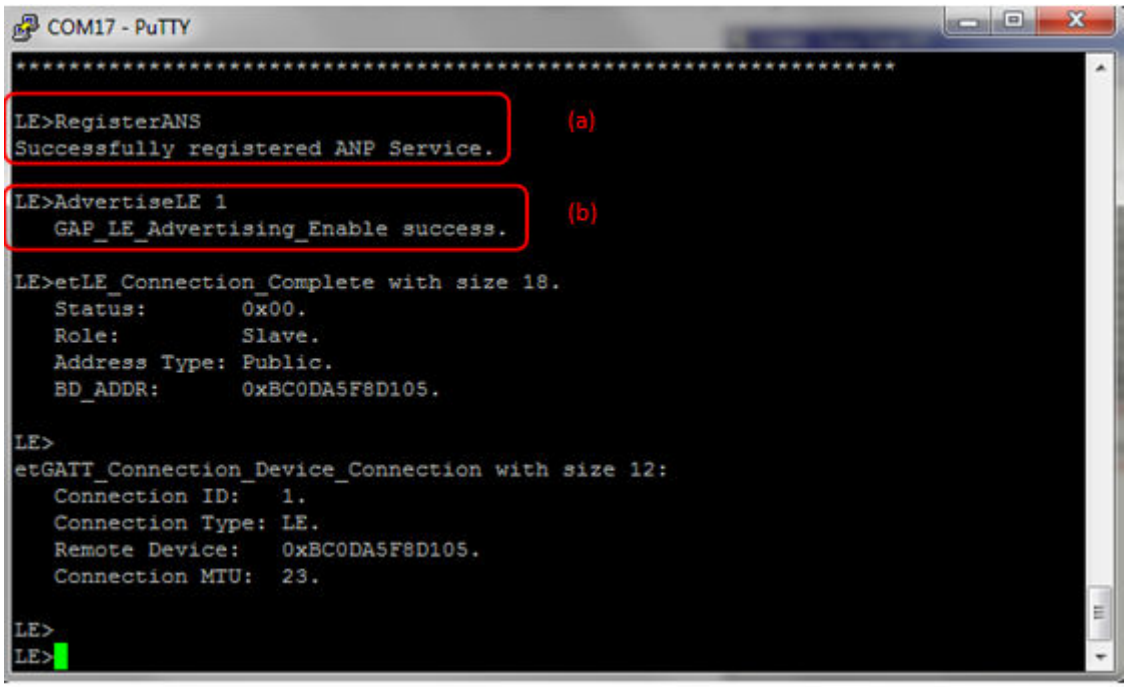

Figure 14-2. ANP Demo ANS Terminal 2

Device 2 (Client) setup on the demo application

Note

Steps c and d are optional if you already know the Bluetooth address of the device that you want to connect to.

1. The Client LE device can try to find which LE devices are in the vicinity using the command: **StartScanning**.
2. Once you have found the device, you can stop scanning by using the command: **StopScanning**.

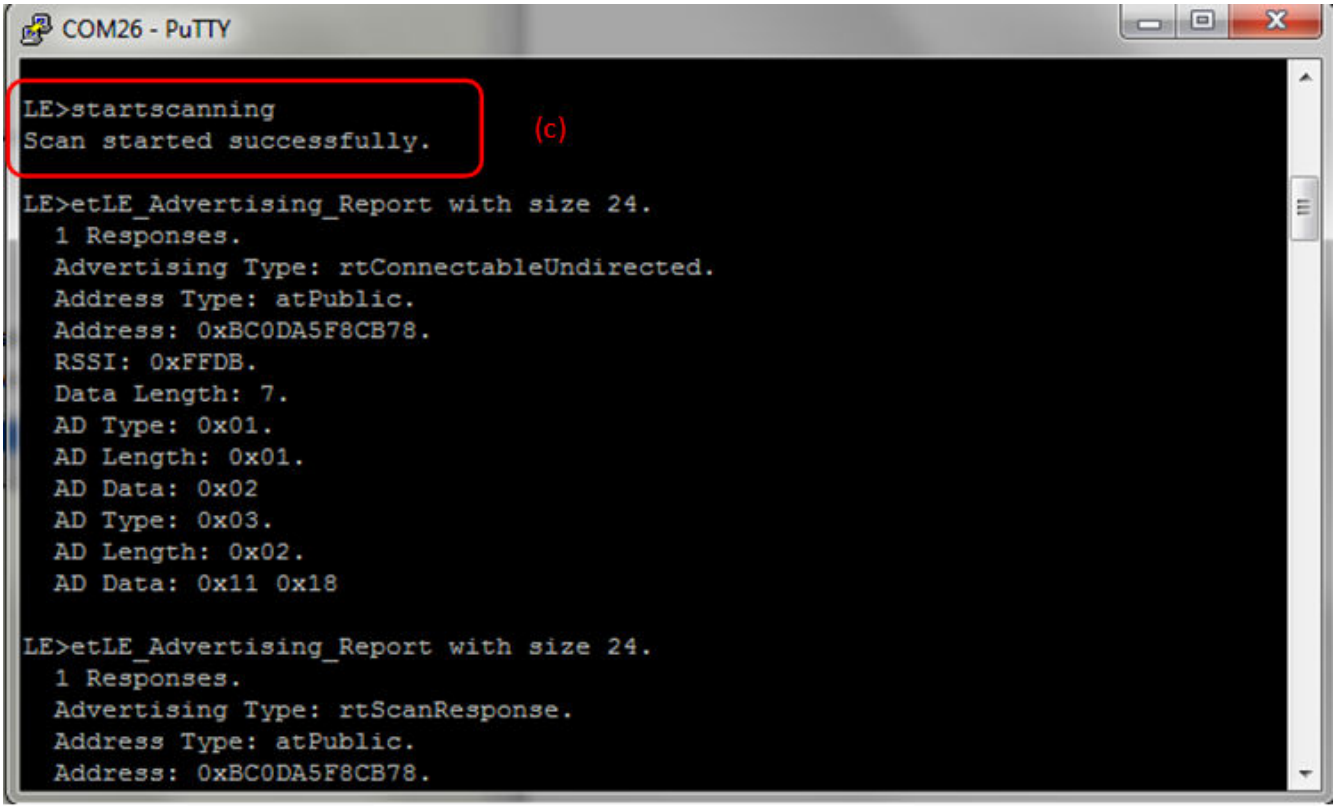

Figure 14-3. ANP Demo ANS Terminal 3

Initiating connection from device 2

1. Once the application on the Client side knows the Bluetooth address of the device that is advertising, it can connect to that device using the command: **ConnectLE <Bluetooth Address>**

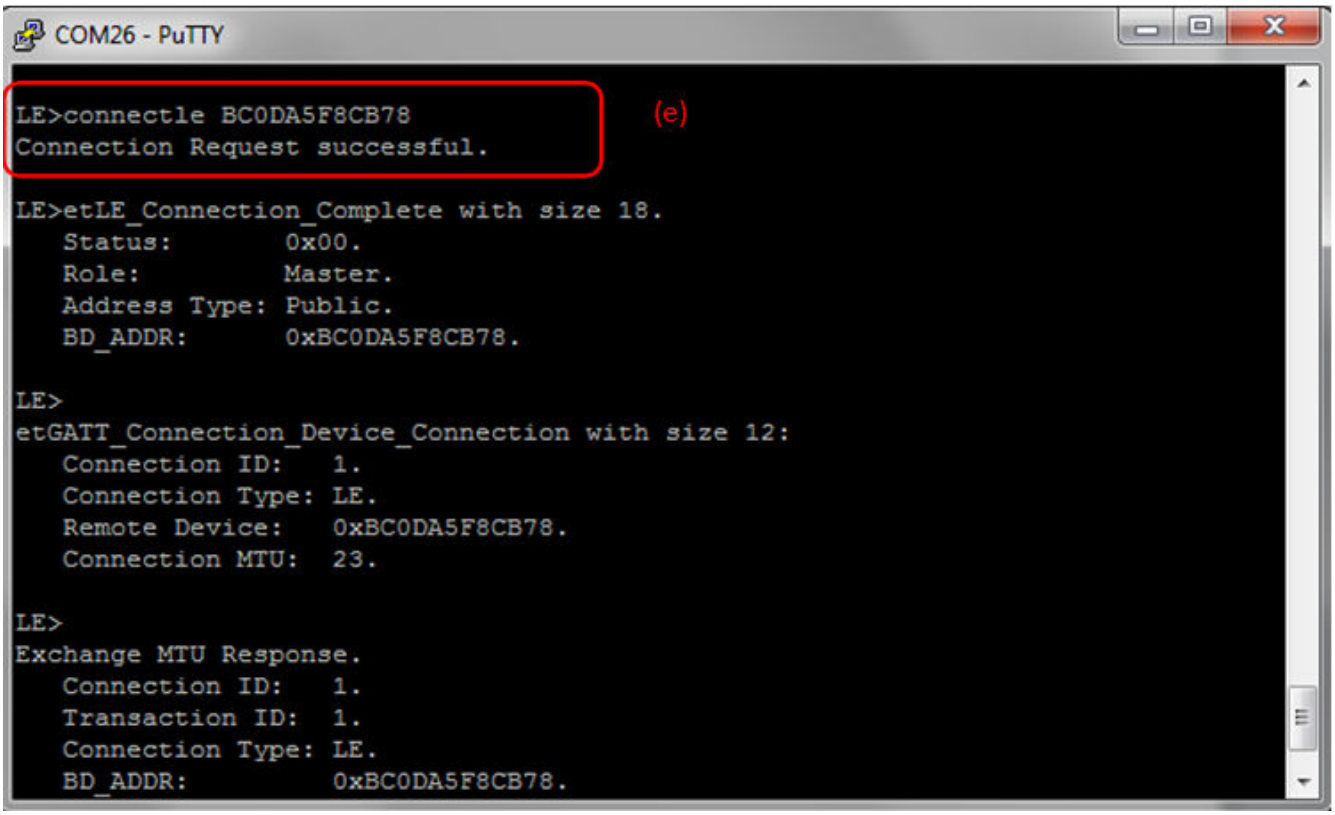

Figure 14-4. ANP Demo ANS Terminal 4

Identify supported services

1. After initialization, the Client needs to find out whether ANS services are supported and what ANS features are available. For this, **DiscoverANS** is run on the Client. After the service discovery operation is complete, the ANS Service Discovery Summary is shown and list of supported features is shown.

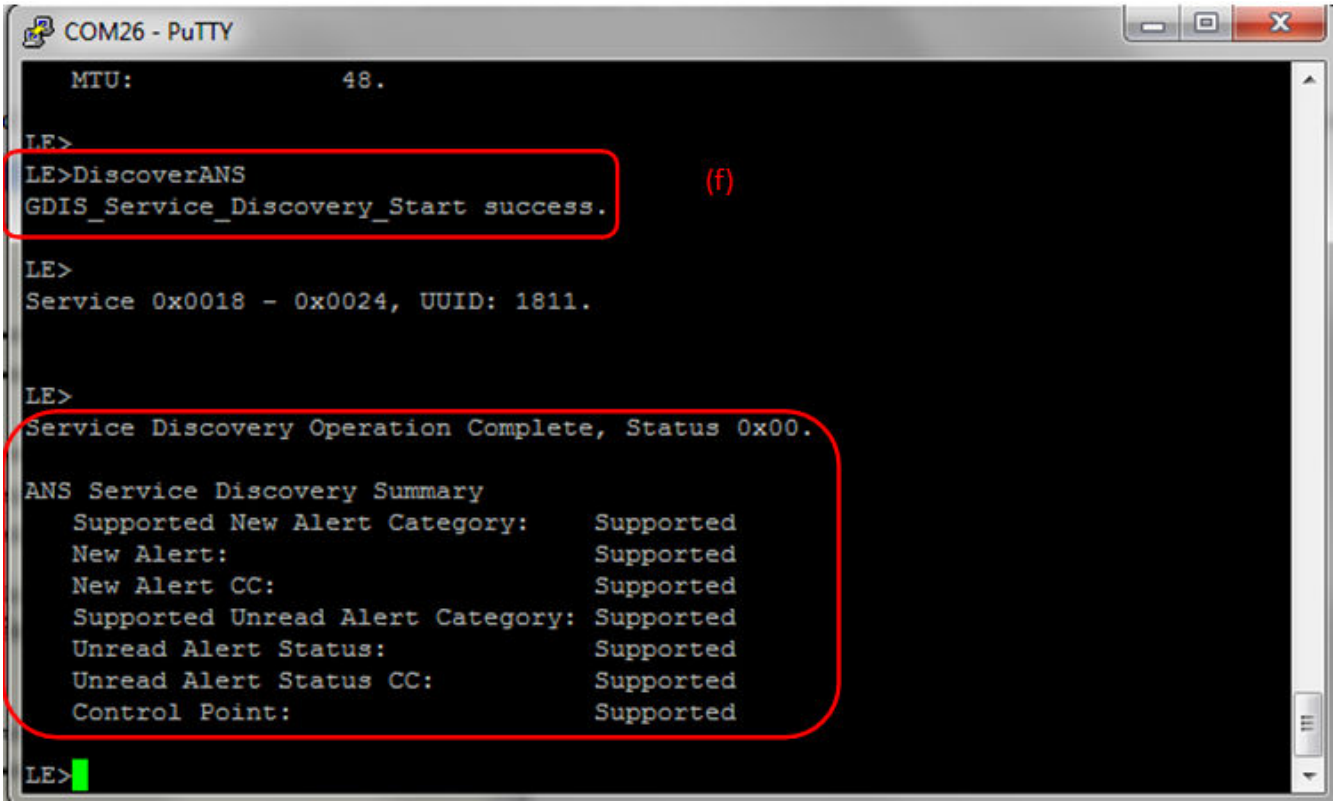

Figure 14-5. ANP Demo ANS Terminal 5

2. After getting the list of supported features, the next step is to configure the ANS on the Client. We can have support for either New Alerts, Unread Alerts or both.

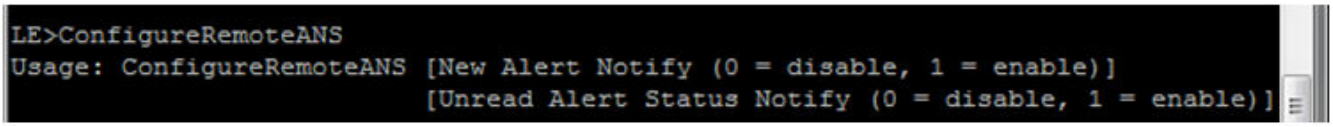

Figure 14-6. ANP Demo ANS Terminal 6

3. In our case, we configure it with both New and Unread Alerts enabled.

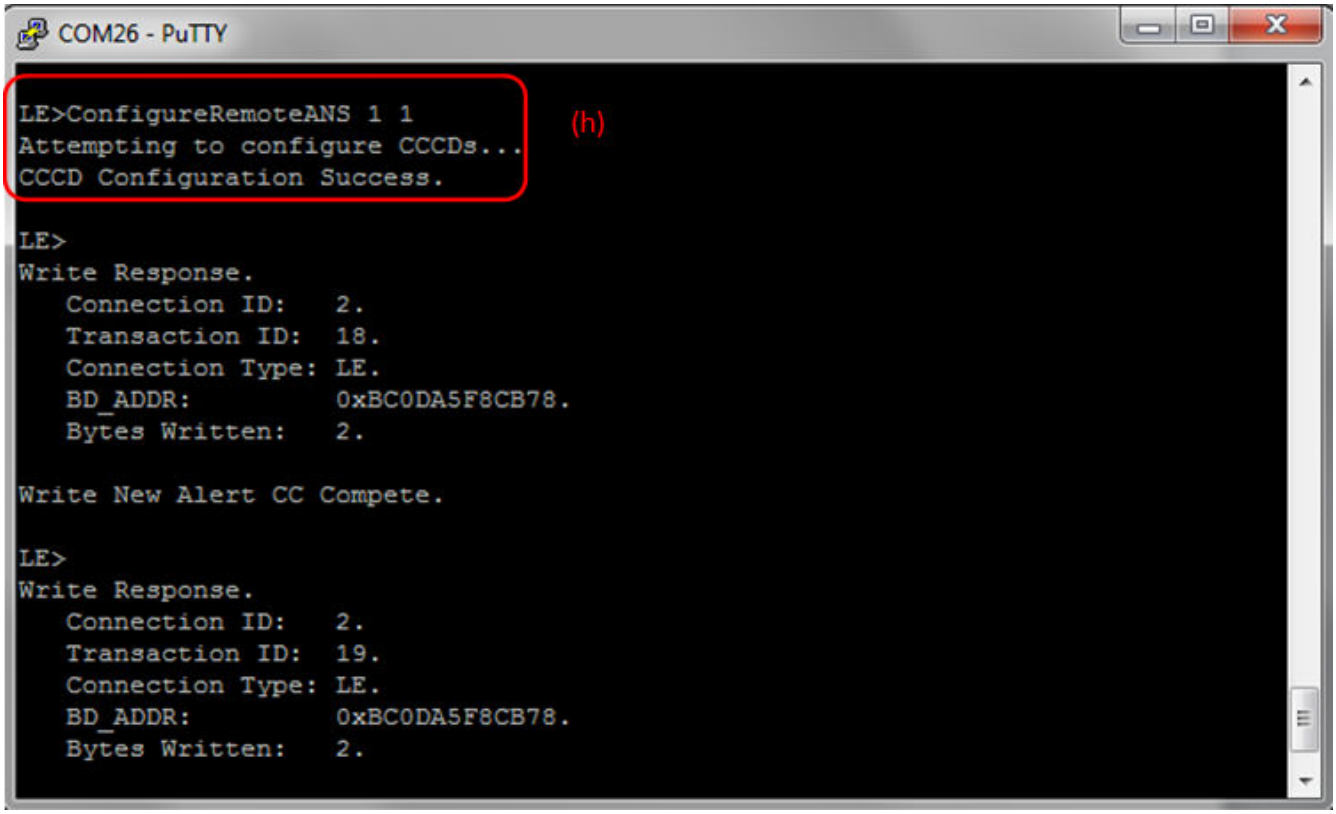

Figure 14-7. ANP Demo ANS Terminal 7

Alert Notification between Client and Server

1. After configuration, the Alert Notification system is active. To send alerts, first the New and Unread alerts need to be enabled on the Client. For this, **EnableNewAlertNotifications** and **EnableUnreadAlertNotifications** is used.
2. For the **EnableNewAlertNotifications** command, the following are the list of options. We can enable either one or all of the options.

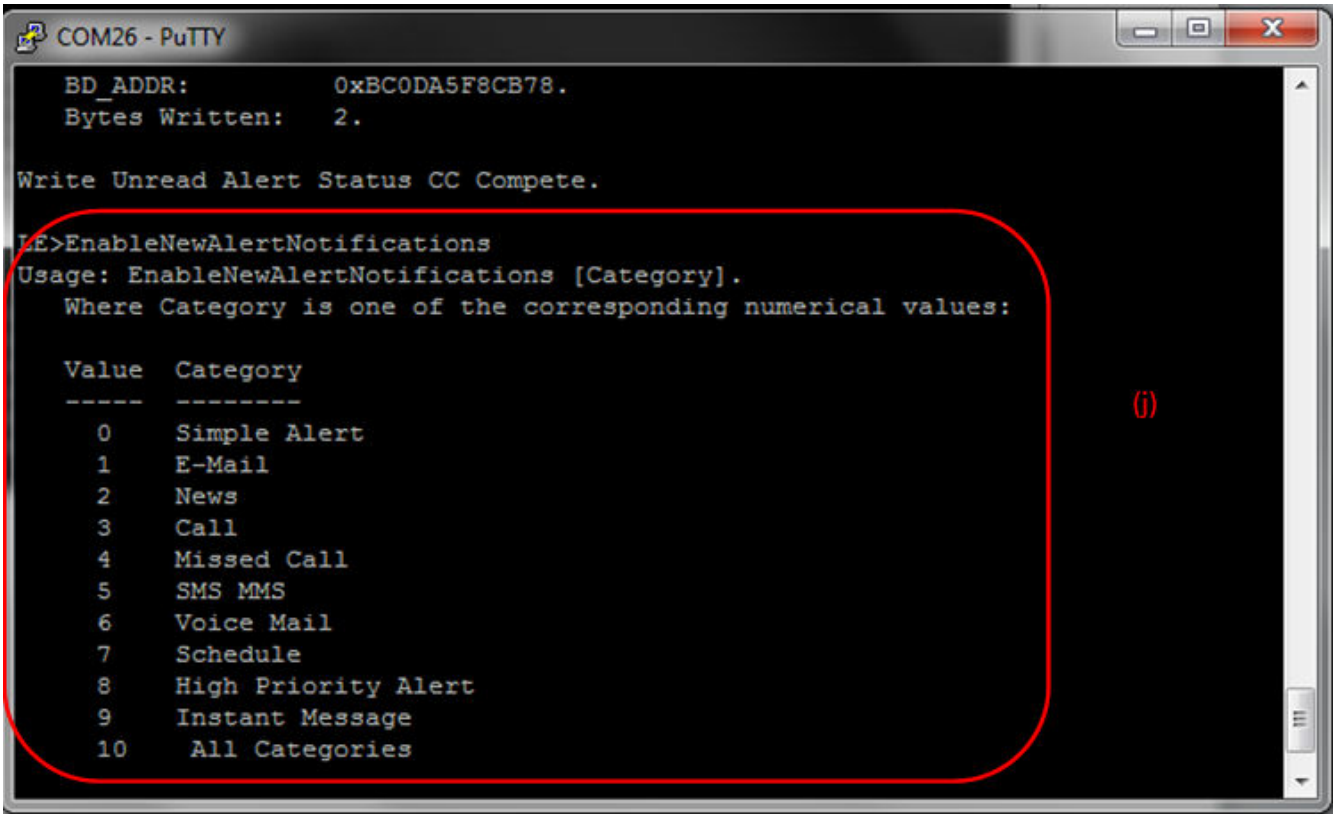

Figure 14-8. ANP Demo ANS Terminal 8

3. In our case, we enable all of them at the same time.

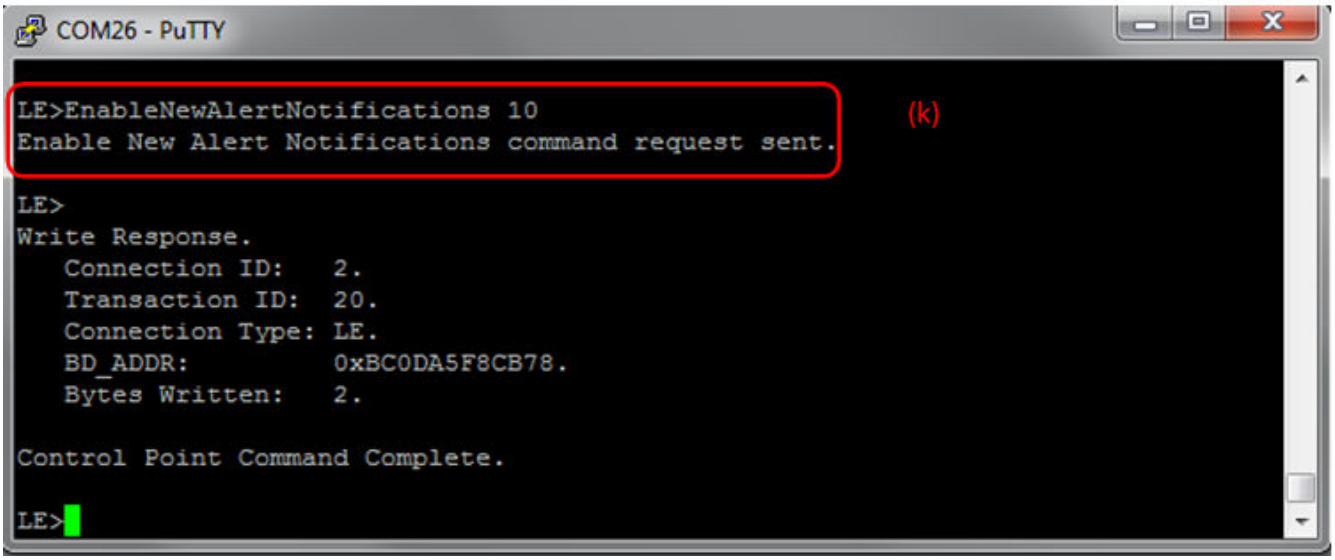

Figure 14-9. ANP Demp ANS Terminal 9

4. Similarly, one can enable either one or all of the options for **EnableUnreadAlertNotifications**. In our case we enable all of them.

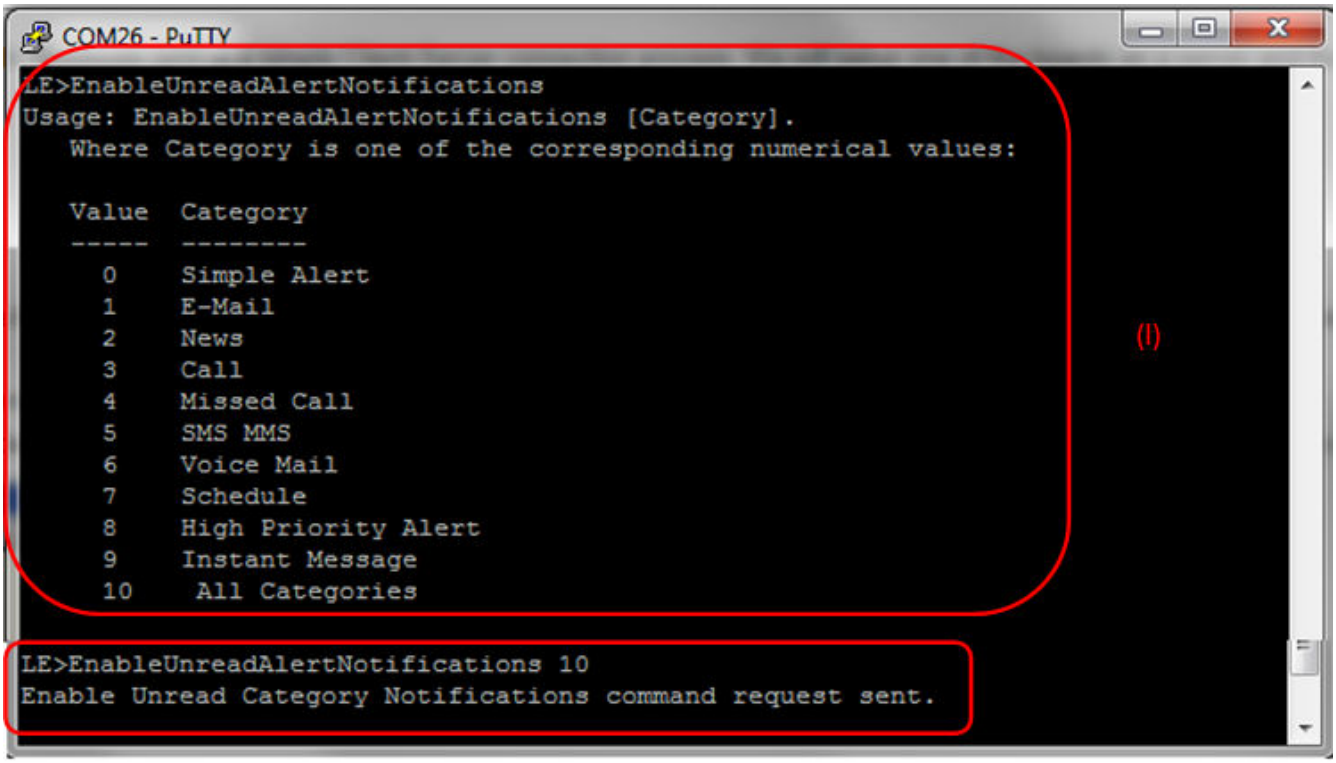

Figure 14-10. ANP Demo ANS Terminal 10

5. After enabling the Alert notifications on the Client side, we can use NotifyNewAlerts to notify the Client from the Server side. NotifyNewAlerts has options similar to the previous menu but the accompanying parameter is a bit field and not a numerical option. For example, to send a Simple Alert we would send **NotifyNewAlerts 1**. To send an Email Alert, we would send **NotifyNewAlerts 2** and to send a New Alert, we would send **NotifyNewAlerts 4**. If we wanted to send both an Email and a Simple Alert we would send **NotifyNewAlerts 3**.

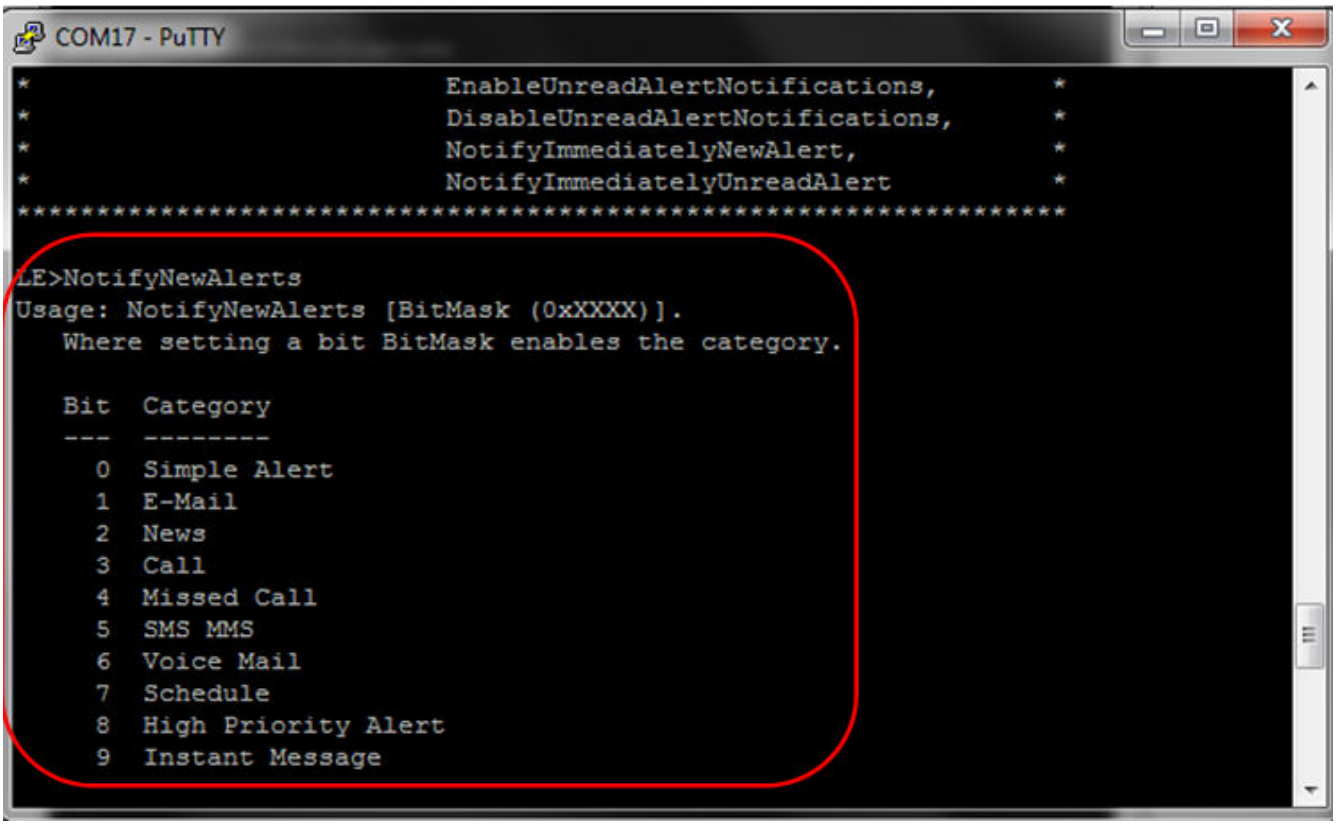

Figure 14-11. ANP Demo ANS Terminal 11

6. In our case we send a simple alert for email. Once it's complete, it shows Success.

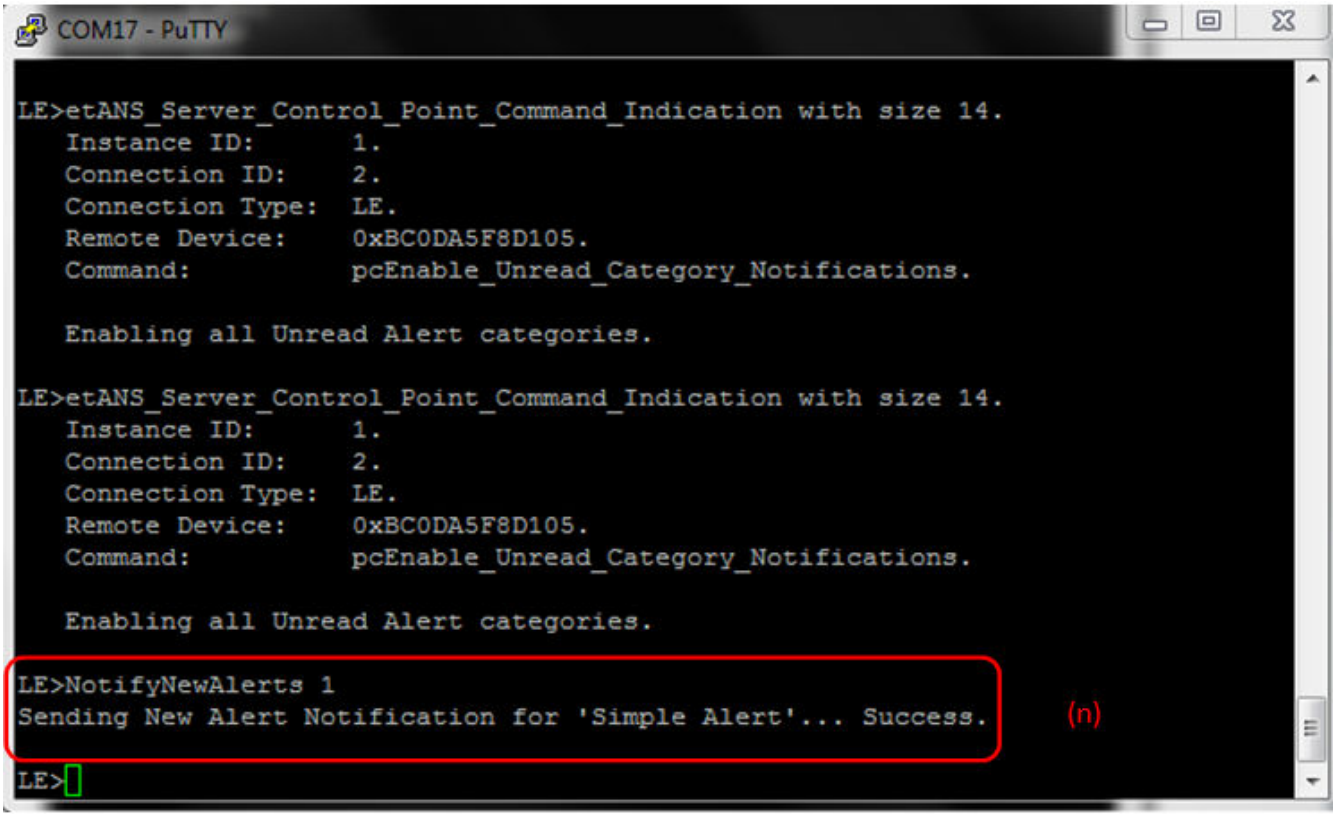

Figure 14-12. ANP Demo ANS Terminal 12

7. On the Client side, it recieves the alert from the sender and it shows New Alert Status with the category of Alert (Simple in our case) and the number of alerts (New Simple Alerts which is zero in our case).

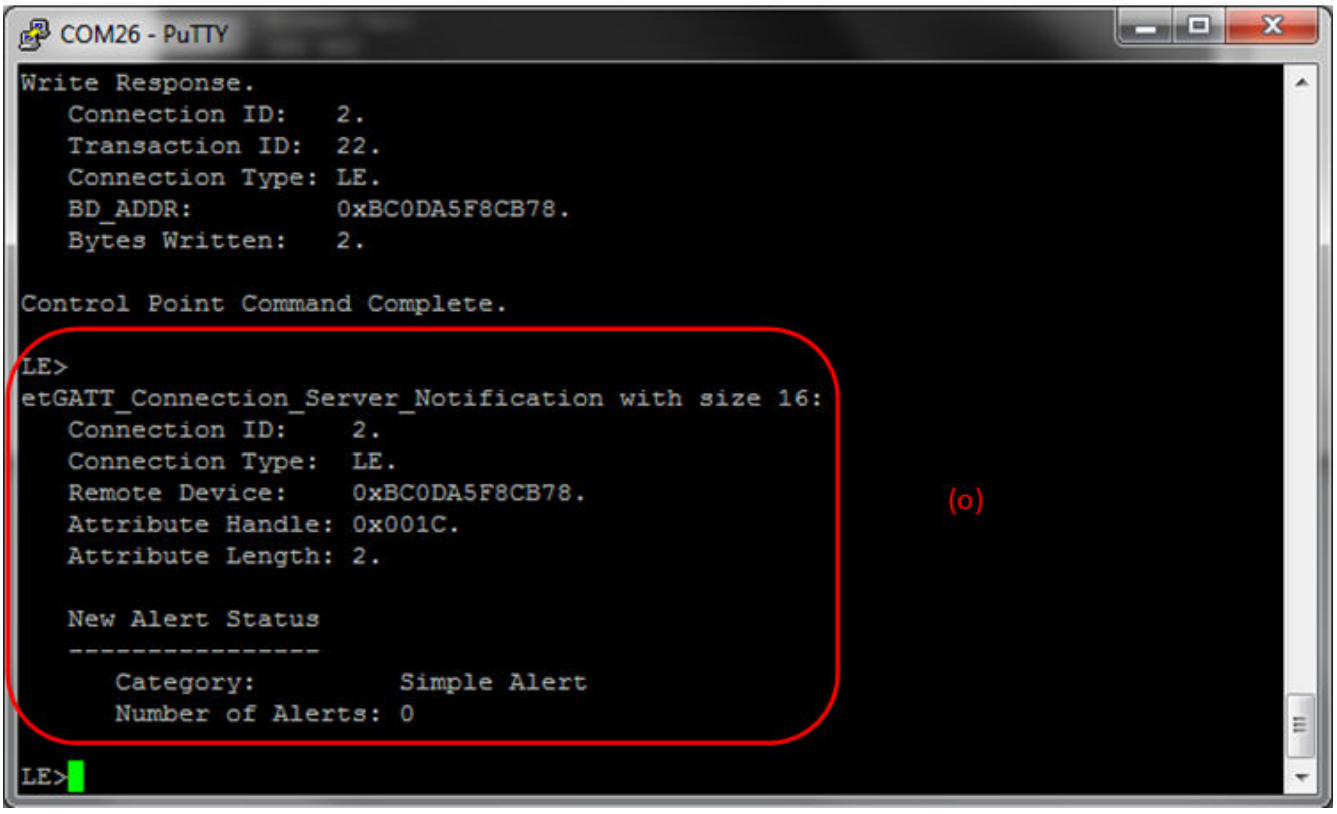

Figure 14-13. ANP Demo ANS Terminal 13

8. NotifyUnreadAlerts like NotifyNewAlerts has the accompanying parameter as a bit field.

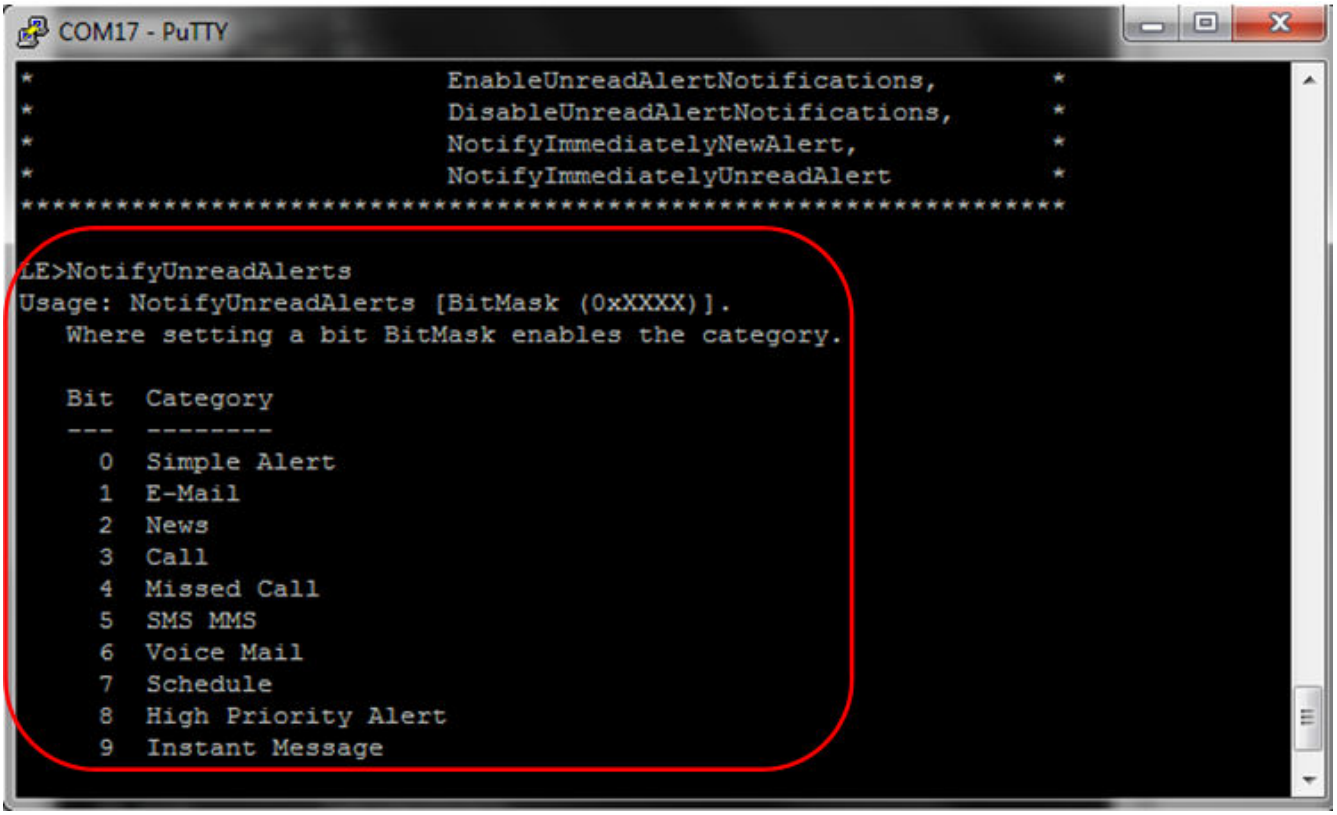

Figure 14-14. ANP Demo ANS Terminal 14

9. In this example case, we send an unread email alert. Once, its transmitted it shows success on the Server side.

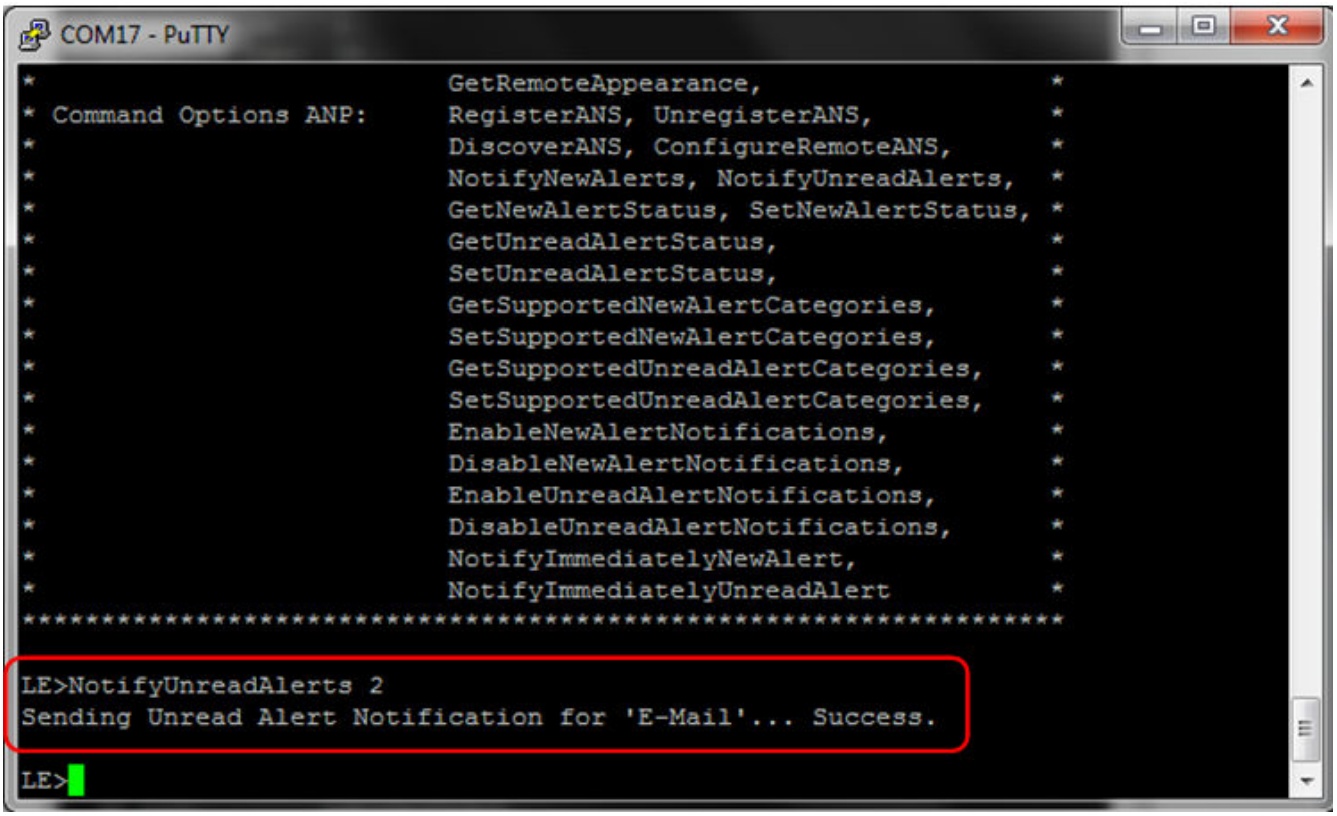

Figure 14-15. ANP Demo ANS Terminal 15

10. Figure 242: ANP Demo ANS Terminal 15 r) And on the Client side, it receives the alert from the sender and it shows Unread Alert Status with the category of Alert (email in our case) and the number of alerts (Unread Emails, which is zero in our case).

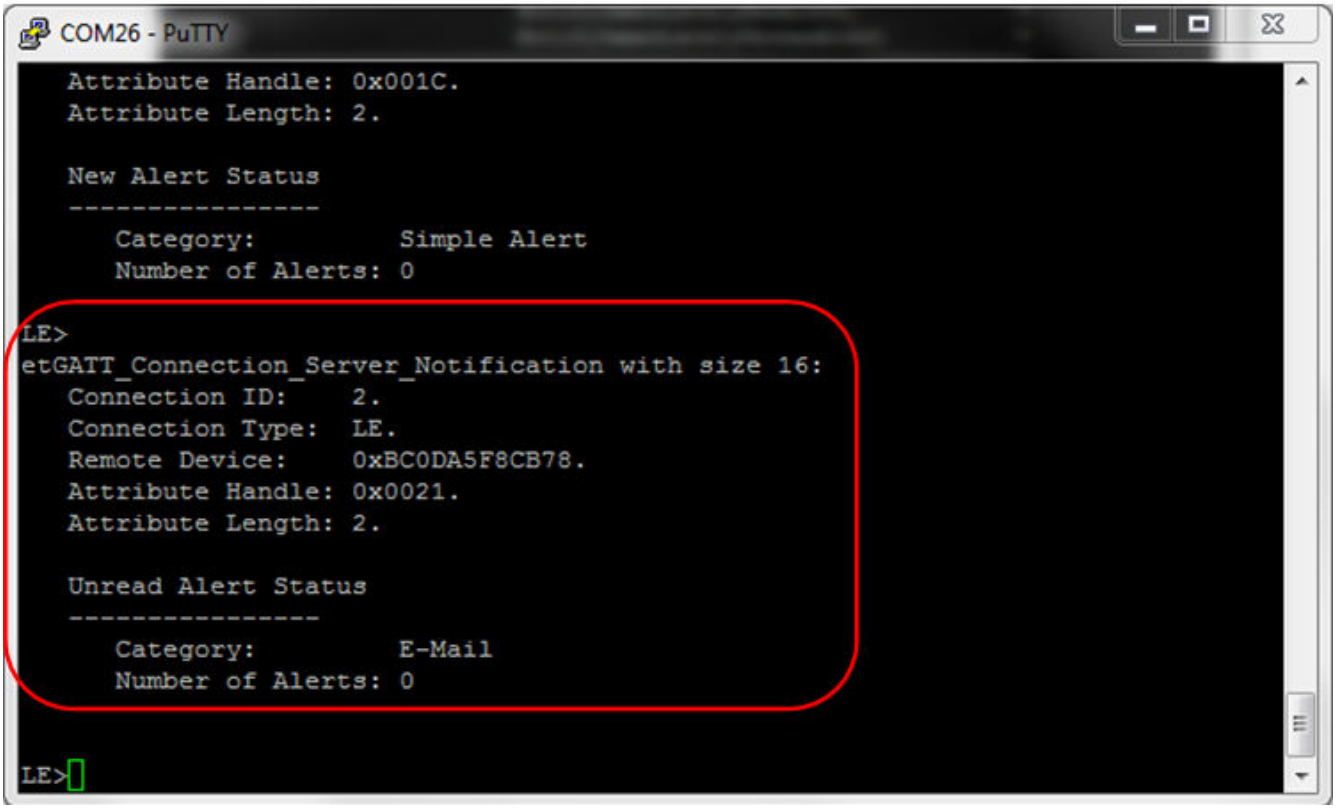

Figure 14-16. ANP Demo ANS Terminal 16

We can also get and set the New and Unread Alerts for the different categories, get and set the Supported categories for New and Unread Alerts, Enable and Disable New and Unread Alert notifications and Notify Immediately New and Unread Alerts.

14.3 Application Commands

TI's Bluetooth stack is implementation of the upper layers of the Bluetooth protocol stack. TI's Bluetooth stackprovides a robust and flexible software development tool that implements the Bluetooth Protocols and Profiles above the Host Controller Interface (HCI). TI's Bluetooth stack's Application Programming Interface (API) provides access to the upper-layer protocols and profiles and can interface directly with the Bluetooth chips.

This page describes the various commands that a user of the application can use. Each command is a wrapper over a TI's Bluetooth stack API which gets invoked with the parameters selected by the user. This is a subset of the APIs available to the user. TI's Bluetooth stack API documentation (**TI_Bluetooth_Stack_Version-Number\Documentation** or for STM32F4, **TI_Bluetooth_Stack_Version-Number\RTOS_VERSION\Documentation**) describes all of the API's in detail.

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands

Alert Notification Profile Commands

RegisterANS

Description

RegisterANS is responsible for registering a ANP Service.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Successfully registered ANP service

(-4) Function_Error

(-1000) ANS_ERROR_INVALID_PARAMETER

(-1001)ANS_ERROR_INSUFFICIENT_RESOURCES

(-1002)ANS_ERROR_SERVICE_ALREADY_REGISTERED

API Call

ANS_Initialize_Service(BluetoothStackID, ANS_EventCallback, NULL, &ANSInstanceID)

API Prototype

int BTPSAPI ANS_Initialize_Service(unsigned int BluetoothStackID, ANS_Event_Callback_t EventCallback, unsigned long CallbackParameter, unsigned int *ServiceID)

Description of API

This function is responsible for opening a ANS Server.The first parameter is the Bluetooth Stack ID on which to open the Server. The second parameter is the Callback function to call when an event occurs on this Server Port. The third parameter is a user-defined callback parameter that will be passed to the callback function with each event. The final parameter is a pointer to store the GATT Service ID of the registered ANS service. This can be used to include the service registered by this call. This function returns the positive, non-zero, Instance ID or a negative error code.

UnRegisterANS

Description

UnRegisterANS is responsible for unregistering a ANP Service.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0)Successfully unregistered ANP service

(-4) Function_Error

(-1000) ANS_ERROR_INVALID_PARAMETER

(-1003) ANS_ERROR_INVALID_INSTANCE_ID

API Call

ANS_Cleanup_Service(BluetoothStackID, ANSInstanceID)

API Prototype

int BTPSAPI ANS_Cleanup_Service(unsigned int BluetoothStackID, unsigned int InstanceID)

Description of API

The following function is responsible for closing a previously opened ANS Server. The first parameter is the Bluetooth Stack ID on which to close the Server. The second parameter is the InstanceID that was returned from a successfull call to ANS_Initialize_Service(). This function returns a zero if successful or a negative return error code if an error occurs.

DiscoverANS

Description

DiscoverANS is responsible for performing a ANP Service Discovery Operation. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0)Successfully unregistered ANP service

(-4) Function_Error

API Call

GDIS_Service_Discovery_Start(BluetoothStackID, ConnectionID, (sizeof(UUID)/sizeof(GATT_UUID_t)), UUID, GDIS_Event_Callback, sdANS)

API Prototype

int BTPSAPI GDIS_Service_Discovery_Start(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GDIS_Event_Callback_t ServiceDiscoveryCallback, unsigned long ServiceDiscoveryCallbackParameter)

Description of API

The GDIS_Service_Discover_Start is in an application module called GDIS that is provided to allow an easy way to perform GATT service discovery. This module can and should be modified for the customers use. This function is called to start a service discovery operation by the GDIS module.

ConfigureRemoteANS

Description

ConfigureRemoteANS is responsible for configure a ANP Service on a remote device. This function will return zero on successful execution and a negative value on errors.

Parameters

There are two parameters for this command, they are [New Alert Notify (0 = disable, 1 = enable)] and [Unread Alert Status Notify (0 = disable, 1 = enable)].

Possible Return Values

(0) CCCD Configuration Success

(-4) FUNCTION_ERROR

(-6) BTPS_ERROR_INVALID_PARAMETER

NotifyNewAlerts

Description

NotifyNewAlerts is responsible for performing a New Alert notification to a connected remote device. This function will return zero on successful execution and a negative value on errors.

Parameters

[BitMask (0xXXXX)]

Category(bit values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message

Possible Return Values

(0) Sending New Alert Notification for 'category'... Success

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

NotifyUnreadAlerts

Description

NotifyUnreadAlerts is responsible for performing an Unread Alert notification to a connected remote device. This function will return zero on successful execution and a negative value on errors.

Parameters

[BitMask (0xXXXX)]

Category(bit values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message

Possible Return Values

(0) Sending Unread Alert Notification for 'category'... Success

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

GetNewAlertStatus

Description

GetNewAlertStatus is responsible for reading the New Alert Status for a specified category. It can be executed only by a Server. This function will return zero on successful execution and a negative value on errors.

Parameters

[Category]

Category(numeric values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message

Possible Return Values

(0) Get New Alert Status request sent

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) ANS_ERROR_INVALID_PARAMETER

(-1003) ANS_ERROR_INVALID_INSTANCE_ID

API Call

ANS_Query_Supported_Categories(BluetoothStackID, ANSInstanceID, scNewAlert, &SupportedCategories))

API Prototype

int BTPSAPI ANS_Query_Supported_Categories(unsigned int BluetoothStackID, unsigned int InstanceID, ANS_Supported_Categories_Type_t SupportedCategoryType, Word_t *SupportedCategoriesMask)

Description of API

The following function is responsible for setting the Alert Notification Supported Categories for the specified Category Type on the specified ANS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to ANS_Initialize_Server(). The next parameter specifies the Category to query the Supported Categories for. The final parameter is a pointer to store the Supported Categories bit mask for the specified ANS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

Note

The SupportedCategoriesMask is a pointer to a bit mask that will be made up of bit masks of the form ANS_SUPPORTED_CATEGORIES_XXX, if this function returns success.

SetNewAlertStatus

Description

SetNewAlertStatus is responsible for writing the New Alert Status for a specified category. It can be executed only by a Server. This function will return zero on successful execution and a negative value on errors.

Parameters

[Category] [Num Alerts] [Alert String (Optional)]

Category(numeric values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message

Possible Return Values

(0) Set New Alert Status success

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) ANS_ERROR_INVALID_PARAMETER

(-1003) ANS_ERROR_INVALID_INSTANCE_ID

API Call

ANS_Query_Supported_Categories(BluetoothStackID, ANSInstanceID, scNewAlert, &SupportedCategories)

API Prototype

int BTPSAPI ANS_Query_Supported_Categories(unsigned int BluetoothStackID, unsigned int InstanceID, ANS_Supported_Categories_Type_t SupportedCategoryType, Word_t *SupportedCategoriesMask)

Description of API

The following function is responsible for setting the Alert Notification Supported Categories for the specified Category Type on the specified ANS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to ANS_Initialize_Server(). The next parameter specifies the Category to query the Supported Categories for. The final parameter is a pointer to store the Supported Categories bit mask for the specified ANS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

Note

The SupportedCategoriesMask is a pointer to a bit mask that will be made up of bit masks of the form ANS_SUPPORTED_CATEGORIES_XXX, if this function returns success.

GetUnreadAlertStatus

Description

GetUnreadAlertStatus is responsible for reading the Unread Alert Status for a specified category. It can be executed only by a Server. This function will return zero on successful execution and a negative value on errors.

Parameters

[Category]

Category(numeric values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message

Possible Return Values

(0) Get Unread Alert Status request sent

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) ANS_ERROR_INVALID_PARAMETER

(-1003) ANS_ERROR_INVALID_INSTANCE_ID

API Call

ANS_Query_Supported_Categories(BluetoothStackID, ANSInstanceID, scUnreadAlertStatus, &SupportedCategories)

API Prototype

int BTPSAPI ANS_Query_Supported_Categories(unsigned int BluetoothStackID, unsigned int InstanceID, ANS_Supported_Categories_Type_t SupportedCategoryType, Word_t *SupportedCategoriesMask)

Description of API

The following function is responsible for setting the Alert Notification Supported Categories for the specified Category Type on the specified ANS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to ANS_Initialize_Server(). The next parameter specifies the Category to query the Supported Categories for. The final parameter is a pointer to store the Supported Categories bit mask for the specified ANS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

Note

The SupportedCategoriesMask is a pointer to a bit mask that will be made up of bit masks of the form ANS_SUPPORTED_CATEGORIES_XXX, if this function returns success.

SetUnreadAlertStatus

Description

SetUnreadAlertStatus is responsible for writing the Unread Alert Status for a specified category. It can be executed only by a Server. This function will return zero on successful execution and a negative value on errors.

Parameters

[Category][NUM VALUES]

Category(numeric values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message

Possible Return Values

(0) Set Unread Alert Status success

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) ANS_ERROR_INVALID_PARAMETER

(-1003) ANS_ERROR_INVALID_INSTANCE_ID

API Call

ANS_Query_Supported_Categories(BluetoothStackID, ANSInstanceID, scUnreadAlertStatus, &SupportedCategories)

API Prototype

int BTPSAPI ANS_Query_Supported_Categories(unsigned int BluetoothStackID, unsigned int InstanceID, ANS_Supported_Categories_Type_t SupportedCategoryType, Word_t *SupportedCategoriesMask)

Description of API

The following function is responsible for setting the Alert Notification Supported Categories for the specified Category Type on the specified ANS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to ANS_Initialize_Server(). The next parameter specifies the Category to query the Supported Categories for. The final parameter is a pointer to store the Supported Categories bit mask for the specified ANS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

Note

The SupportedCategoriesMask is a pointer to a bit mask that will be made up of bit masks of the form ANS_SUPPORTED_CATEGORIES_XXX, if this function returns success.

GetSupportedNewAlertCategories

Description

GetSupportedNewAlertCategories is responsible for reading the Supported New Alert Category. It can be executed by a Server or a Client with an open connection to a remote Server. If executed as a Client, a GATT read request will be generated, and the results will be returned as a response in the GATT Client event callback. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Get Supported New Alert Category Request sent, Transaction ID = %u

(-4) FUNCTION_ERROR

(-1000) ANS_ERROR_INVALID_PARAMETER

(-1003) ANS_ERROR_INVALID_INSTANCE_ID

API Call

ANS_Query_Supported_Categories(BluetoothStackID, ANSInstanceID, scNewAlert, &SupportedCategories)

API Prototype

int BTPSAPI ANS_Query_Supported_Categories(unsigned int BluetoothStackID, unsigned int InstanceID, ANS_Supported_Categories_Type_t SupportedCategoryType, Word_t *SupportedCategoriesMask)

Description of API

The following function is responsible for setting the Alert Notification Supported Categories for the specified Category Type on the specified ANS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to ANS_Initialize_Server(). The next parameter specifies the Category to query the Supported Categories for. The final parameter is a pointer to store the Supported Categories bit mask for the specified ANS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

Note

The SupportedCategoriesMask is a pointer to a bit mask that will be made up of bit masks of the form ANS_SUPPORTED_CATEGORIES_XXX, if this function returns success.

SetSupportedNewAlertCategories

Description

SetSupportedNewAlertCategories is responsible for writing the Supported New Alert Category. It can be executed only by a Server. This function will return zero on successful execution and a negative value on errors.

Parameters

[BitMask (0xXXXX)]

Category(bit values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message

Possible Return Values

(0) Supported New Alert Category successfully set

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-1000) ANS_ERROR_INVALID_PARAMETER

(-1003) ANS_ERROR_INVALID_INSTANCE_ID

API Call

ANS_Set_Supported_Categories(BluetoothStackID, ANSInstanceID, scNewAlert, SupportedCategories)

API Prototype

int BTPSAPI ANS_Set_Supported_Categories(unsigned int BluetoothStackID, unsigned int InstanceID, ANS_Supported_Categories_Type_t SupportedCategoryType, Word_t SupportedCategoriesMask)

Description of API

The following function is responsible for setting the Alert Notification Supported Categories for the specified Category Type on the specified ANS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to ANS_Initialize_Server(). The next parameter specifies the Category to set the Supported Categories for. The final parameter is the Supported Categories bit mask to set as the supported categories for the specified ANS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

Note

The SupportedCategoriesMask is a bit mask that is made up of bit masks of the form ANS_SUPPORTED_CATEGORIES_XXX.

GetSupportedUnreadAlertCategories

Description

GetSupportedUnreadAlertCategories is responsible for reading the Supported Unread Alert Category. It can be executed by a Server or a Client with an open connection to a remote Server. If executed as a Client, a GATT read request will be generated, and the results will be returned as a response in the GATT Client event callback. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Get Supported Unread Alert Category Request sent, Transaction ID = %u

(-4) FUNCTION_ERROR

(-1000) ANS_ERROR_INVALID_PARAMETER

(-1003) ANS_ERROR_INVALID_INSTANCE_ID

API Call

ANS_Query_Supported_Categories(BluetoothStackID, ANSInstanceID, scUnreadAlertStatus, &SupportedCategories)

API Prototype

int BTPSAPI ANS_Query_Supported_Categories(unsigned int BluetoothStackID, unsigned int InstanceID, ANS_Supported_Categories_Type_t SupportedCategoryType, Word_t *SupportedCategoriesMask)

Description of API

The following function is responsible for setting the Alert Notification Supported Categories for the specified Category Type on the specified ANS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to ANS_Initialize_Server(). The next parameter specifies the Category to query the Supported Categories for. The final parameter is a pointer to store the Supported Categories bit mask for the specified ANS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

Note

The SupportedCategoriesMask is a pointer to a bit mask that will be made up of bit masks of the form ANS_SUPPORTED_CATEGORIES_XXX, if this function returns success.

SetSupportedUnreadAlertCategories

Description

SetSupportedUnreadAlertCategories is responsible for writing the Supported Unread Alert Category. It can be executed only by a Server. This function will return zero on successful execution and a negative value on errors.

Parameters

[BitMask (0xXXXX)]

Category(bit values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message

Possible Return Values

(0) Supported Unread Alert Category successfully set

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-1000) ANS_ERROR_INVALID_PARAMETER

(-1003) ANS_ERROR_INVALID_INSTANCE_ID

API Call

ANS_Set_Supported_Categories(BluetoothStackID, ANSInstanceID, scUnreadAlertStatus, SupportedCategories)

API Prototype

int BTPSAPI ANS_Set_Supported_Categories(unsigned int BluetoothStackID, unsigned int InstanceID, ANS_Supported_Categories_Type_t SupportedCategoryType, Word_t SupportedCategoriesMask)

Description of API

The following function is responsible for setting the Alert Notification Supported Categories for the specified Category Type on the specified ANS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to ANS_Initialize_Server(). The next parameter specifies the Category to set the Supported Categories for. The final parameter is the Supported Categories bit mask to set as the supported categories for the specified ANS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

Note

The SupportedCategoriesMask is a bit mask that is made up of bit masks of the form ANS_SUPPORTED_CATEGORIES_XXX.

EnableNewAlertNotifications

Description

EnableNewAlertNotifications is responsible for writing the Enable New Incoming Alert Notifications command to a remote Server Control Point. It can be executed only by a Client. This function will return zero on successful execution and a negative value on errors.

Parameters

[Category]

Category(numeric values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message, 10 = All Categories

Possible Return Values

(0) Enable New Alert Notifications command request sent.

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

DisableNewAlertNotifications

Description

DisableNewAlertNotifications is responsible for writing the Disable New Incoming Alert Notifications command to a remote Server Control Point. It can be executed only by a Client. This function will return zero on successful execution and a negative value on errors.

Parameters

[Category]

Category(numeric values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message, 10 = All Categories

Possible Return Values

(0) Disable New Alert Notifications command request sent

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

EnableUnreadAlertNotifications

Description

EnableUnreadAlertNotifications is responsible for writing the Enable Unread Category Status Notifications command to a remote Server Control Point. It can be executed only by a Client. This function will return zero on successful execution and a negative value on errors.

Parameters

[Category]

Category(numeric values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message, 10 = All Categories

Possible Return Values

(0) Enable Unread Category Notifications command request sent

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

DisableUnreadAlertNotifications

Description

DisableUnreadAlertNotifications is responsible for writing the Disable Unread Category Status Notifications command to a remote Server Control Point. It can be executed only by a Client. This function will return zero on successful execution and a negative value on errors.

Parameters

[Category]

Category(numeric values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message, 10 = All Categories

Possible Return Values

(0) Disable Unread Category Notifications command request sent

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

NotifyImmediatelyNewAlerts

Description

NotifyImmediatelyNewAlerts is responsible for writing the Notify New Incoming Alert Immediately command to a remote Server Control Point. It can be executed only by a Client. This function will return zero on successful execution and a negative value on errors.

Parameters

[Category]

Category(numeric values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message, 10 = All Categories

Possible Return Values

(0) Notify New Alert Immediately command request sent

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

NotifyImmediatelyUnreadAlerts

Description

NotifyImmediatelyUnreadAlerts is responsible for writing the Notify New Unread Category Status Immediately command to a remote Server Control Point. It can be executed only by a Client. This function will return zero on successful execution and a negative value on errors.

Parameters

[Category]

Category(numeric values): 0 = Simple Alert, 1 = E-Mail, 2 = News, 3 = Call, 4 = Missed Call, 5 = SMS MMS, 6 = Voice Mail, 7 = Schedule, 8 = High Priority Alert, 9 = Instant Message, 10 = All Categories

Possible Return Values

(0) Notify Unread Category Immediately command request sent

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

15 HFP Demo Guide

15.1 Demo Overview

Note

The same instructions can be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms.

The Heart Rate profile (HRP) enables a Collector device to connect and interact with a Heart Rate sensor for use in healthcare applications. There are two roles defined in this profile. The first is the Sensor which measures the Heart Rate and the second is the collector which gets the Heart Rate and other settings from the sensor. Typically, the sensor would be present directly on the patient in a location such as the Heart or wrist measuring the temperature while the collector device is close by getting the Heart Rate from the sensor at regular intervals.

This application allows the user to use a console to use Bluetooth Low Energy (BLE) to establish connection between two BLE devices, notify the heart rate between the service and Client, get and change the location of the heart rate sensor.

It is recommended that the user visits the kit setup Getting Started Guide for MSP430, Getting Started Guide for TIVA, Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as MSP-EXP430F5438 USB - Serial Port (COM x), Tiva Virtual COM Port (COM x), XDS110 Class Application/User UART (COM x) for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud (9600 for MSP430), 8, n, 1. Once connected, reset the device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

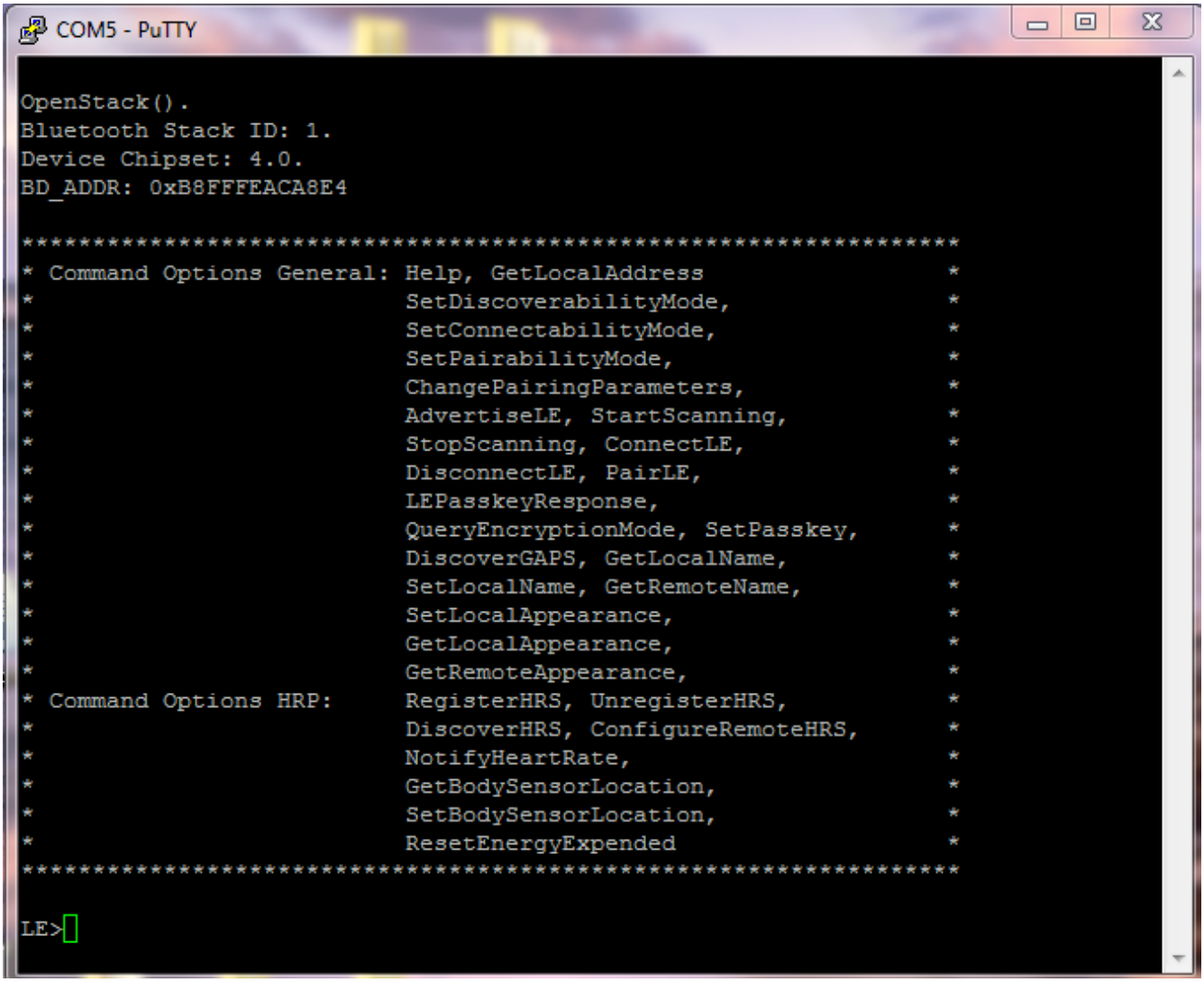

Figure 15-1. HFP Demo Start

Now connect the second board via miniUSB or microUSB cable and follow the same steps performed before within the Running the Bluetooth Code section on the first board. The second device that is connected to the computer will be the Client.

15.2 Demo Application

This section provides a description of how to use the demo application to connect two configured boards and communicate over Bluetooth. The Bluetooth Heart Rate Service (HRS) is a simple Client-Server connection process. We will setup one of the boards as a Server and the other board as a Client. We will then initiate a connection from the Client to the Server. Once connected, we can transmit data between the two devices over Bluetooth.

Server setup on the demo application

1. We will setup the first board as a Server. Note the Bluetooth address of the Server; we will later use this to initiate a connection from the Client.
2. Two commands are all that is needed to setup the Server. The first is RegisterHRS, so issue the RegisterHRS 1 command.
3. Now use the AdvertiseLE command by issuing the AdvertiseLE 1 command.

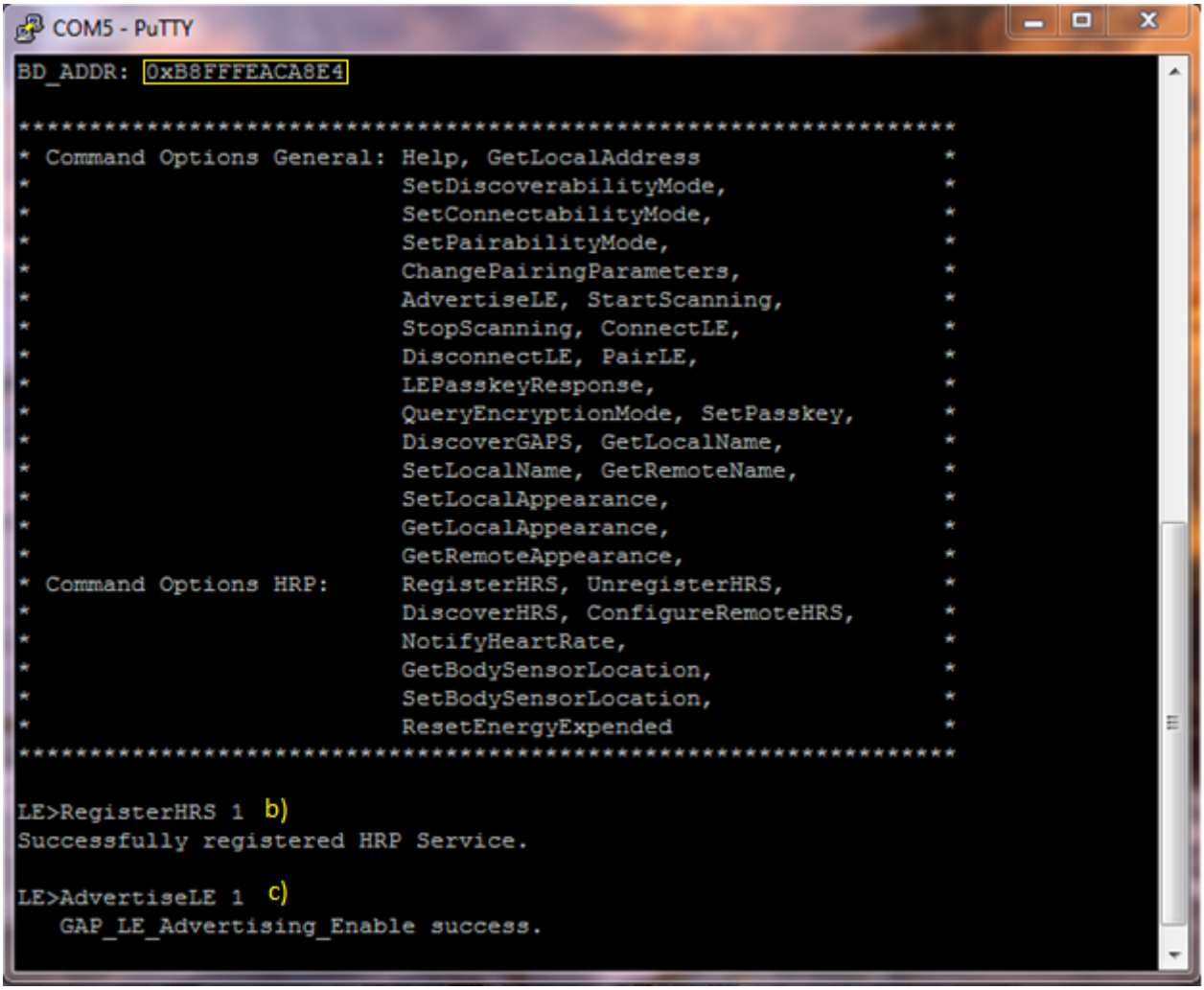

Figure 15-2. HRP Demo Register HRS and Advertise

Initiating connection from the Client

Note

Steps 1 and 2 are optional if you already know the Bluetooth address of the device that you want to connect to.

1. The Client LE device can try to find which LE devices are in the vicinity issuing the command: StartScanning.
2. Once you have found the device, you can stop scanning by issuing the command: StopScanning

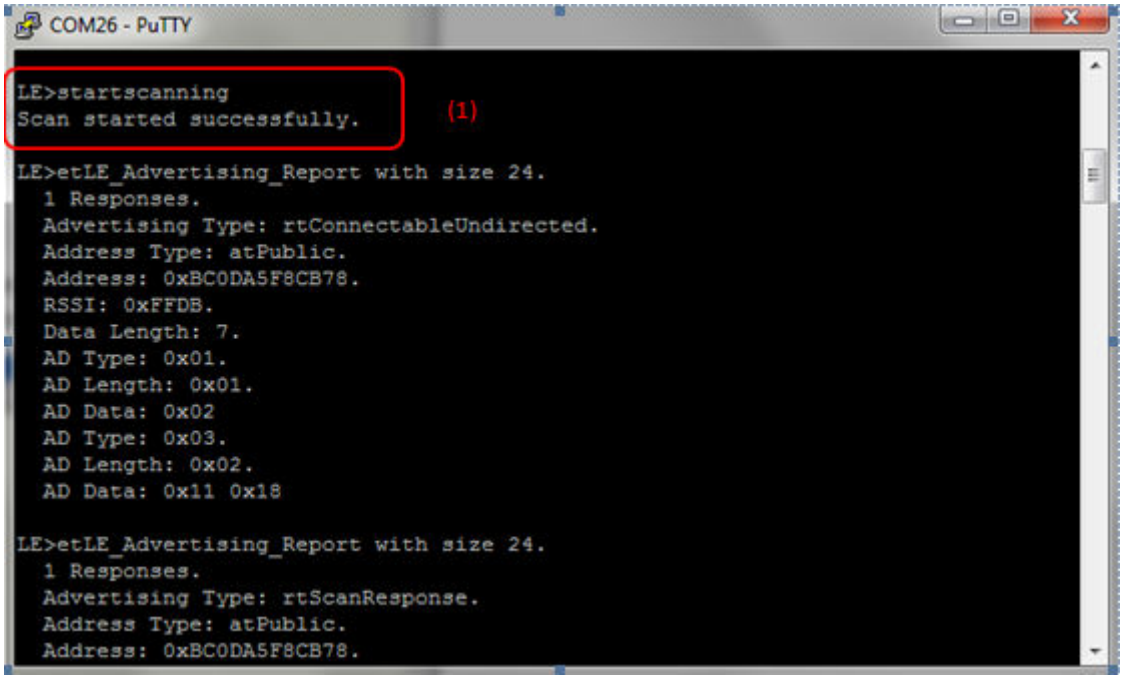

Figure 15-3. HRP Demo Scanning

3. Retrieve the Bluetooth address of the first board that was configured as a Server.
4. Issue a ConnectLE <BD_ADDR of Server> command in the Client terminal.
5. When a Client successfully connects to a Server, both the Client and Server will output LE_Connection_Complete and information about the current connection.

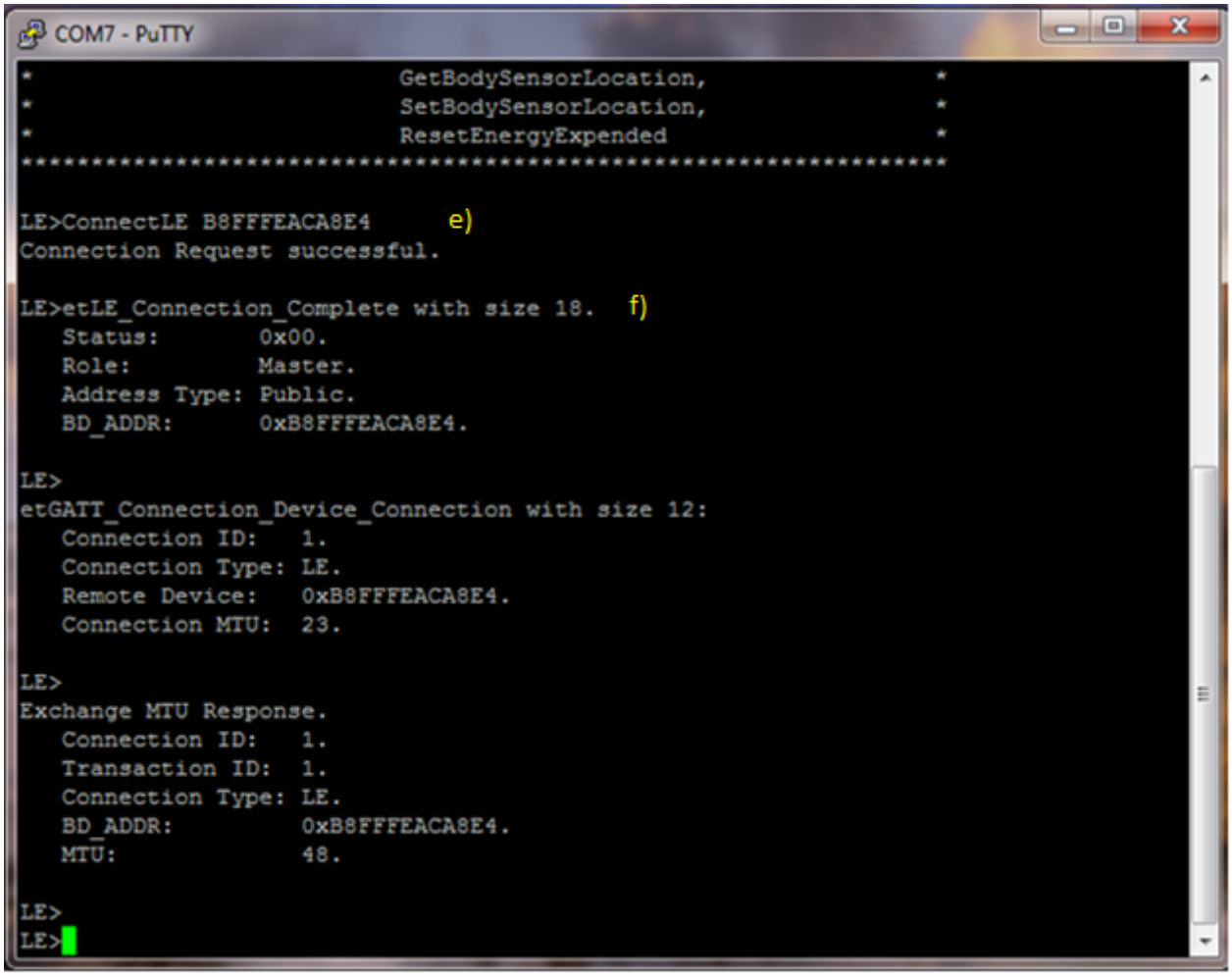

Figure 15-4. HRP Demo Connect LE

Sending Heart Rate Information between Client and Server

1. Now we have a connection established and both devices are ready to send data to each other.
2. Before heart rate information can be sent we must first initialize commands in the Client terminal.
3. The two commands are DiscoverHRS and ConfigureRemoteHRS, so issue the command DiscoverHRS 1. A list of supported services should be displayed.
4. Now issue the ConfigureRemoteHRS 1 command. If the configuration succeeds information about the CCCD configuration will be displayed in the Client and Server terminals.

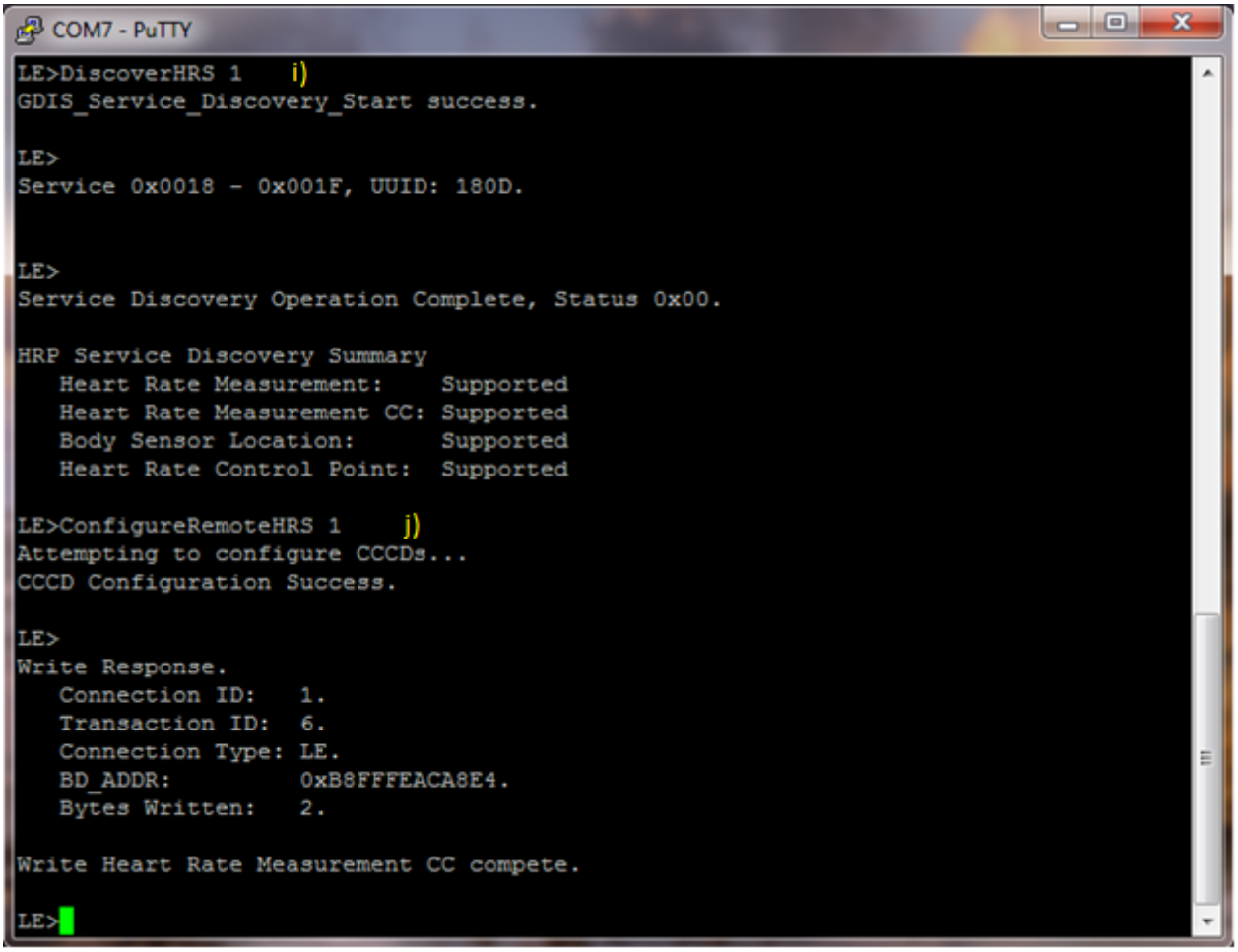

Figure 15-5. HRP Demo Discover HRS

5. To send information about the heart rate, use the NotifyHeartRate command in the Server terminal. The parameters for the NotifyHeartRate command are [HR (BPM)] [HR Format (0 = Byte, 1 = Word)] [Sensor Contact Status (0 = Not Supported, 1 = Supported/Not Detected, 2 = Supported/Detected] [Energy Expended (0 = Don't send, 1 = Send)] [RR Intervals (0 = None, X = Number of Intervals)]. Use the parameters 80 1 2 1 0 after the command.
6. Information about the heart rate should now be sent to the Client and be displayed in the Client terminal.

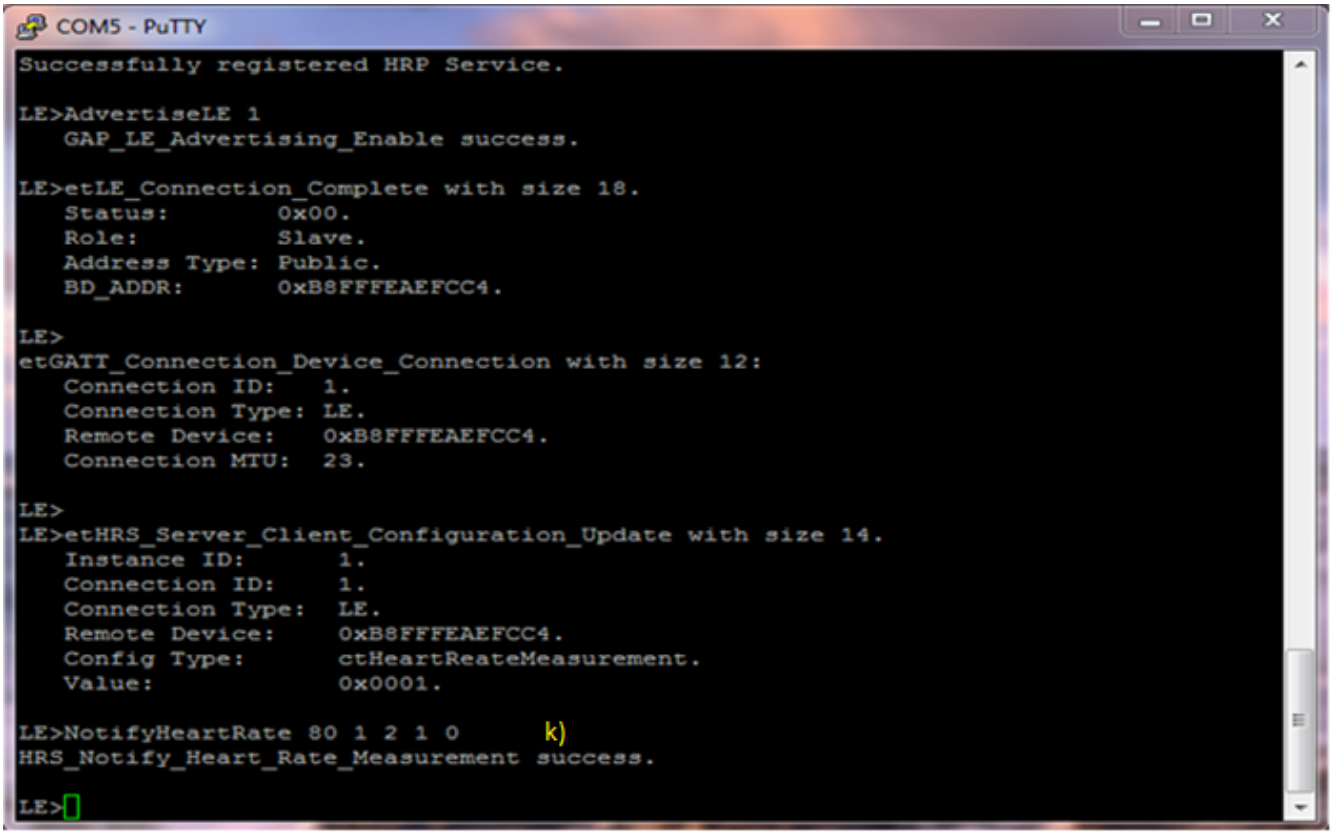

Figure 15-6. HRP Demo Notify Heart Rate

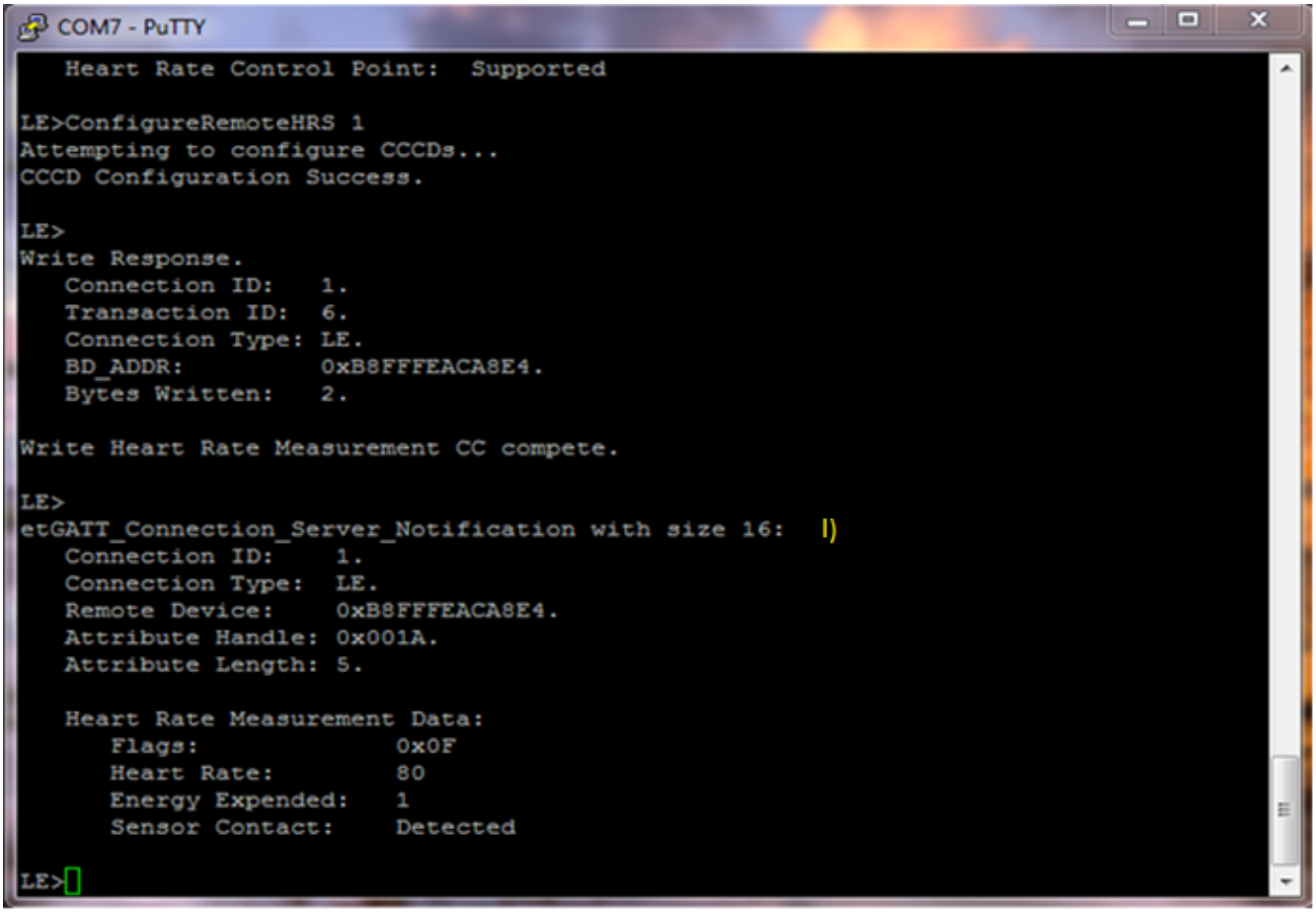

Figure 15-7. HRP Demo Heart Rate Measurement

Setting a local appearance on the Server and checking that appearance from the Client

1. In the Server terminal set the type of device with the command SetLocalAppearance and the parameter [index]. There are over 20 types of devices in the index, to look at them type first enter only SetLocalAppearance.
2. Now use the DiscoverGAPS command in the Client terminal.
3. Lastly enter GetRemoteAppearance in the Client terminal to see what type of device the Server had been set to.

Setting the location of the heart rate sensor on the Server and reading it from the Client

1. In the Server terminal set the location of the heart rate sensor with the command SetBodySensorLocation [type]. Type = 0 – Other, 1 – Chest, 2 – Wrist, 3 – Finger, 4 – Hand, 5 - Ear Lobe, 6 – Foot.
2. To see the location of the heart rate sensor type GetBodySensorLocation in the Client terminal.

15.3 Application Commands

RegisterHRS

Description

the RegisterHRS command is responsible for registering a HRP Service. This function will return zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Successfully registered an HRP service

(-4) FUNCTION_ERROR

(-1000) HRS_ERROR_INVALID_PARAMETER

(-1002) HRS_ERROR_INSUFFICIENT_RESOURCES

(-1003) HRS_ERROR_SERVICE_ALREADY_REGISTERED

API Call

HRS_Initialize_Service(BluetoothStackID, HRS_HEART_RATE_CONTROL_POINT_RESET_ENERGY_EXPENDED_SUPPORTED, HRS_EventCallback, NULL, &HRSInstanceID);

API Prototype

int BTPSAPI HRS_Initialize_Service(unsigned int BluetoothStackID, unsigned long Supported_Commands, HRS_Event_Callback_t EventCallback, unsigned long CallbackParameter, unsigned int *ServiceID)

Description of API

This function is responsible for opening a HRS Server. The first parameter is the Bluetooth Stack ID on which to open the Server. The second parameter is the mask of supported Heart Rate Control point commands. The third parameter is the Callback function to call when an event occurs on this Server Port. The fourth parameter is a user-defined callback parameter that will be passed to the callback function with each event. The final parameter is a pointer to store the GATT Service ID of the registered HRS service. This can be used to include the service registered by this call. This function returns the positive, non-zero, Instance ID or a negative error code. Only 1 HRS Server may be open at a time, per Bluetooth Stack ID. The Supported_Commands parameter must be made up of bit masks of the form: HRS_HEART_RATE_CONTROL_POINT_XXX_SUPPORTED All Client Requests will be dispatch to the EventCallback function that is specified by the second parameter to this function.

UnRegisterHRS

Description

The UnRegisterHRS command is responsible for unregistering a HRP Service. This command will return zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Successfully closed the HRS Server

(-4) FUNCTION_ERROR (HRP Service not registered)

(-1000) HRS_ERROR_INVALID_PARAMETER

(-1004) HRS_ERROR_INVALID_INSTANCE_ID

API Call

HRS_Cleanup_Service(BluetoothStackID, HRSInstanceID);

API Prototype

int BTPSAPI HRS_Cleanup_Service(unsigned int BluetoothStackID, unsigned int InstanceID)

Description of API

This function is responsible for closing a previously opened HRS Server. The first parameter is the Bluetooth Stack ID on which to close the Server. The second parameter is the InstanceID that was returned from a successful call to HRS_Initialize_Service(). This function returns a zero if successful or a negative return error code if an error occurs.

DiscoverHRS

Description

The DiscoverHRS command is responsible for performing a HRP Service Discovery Operation. This command will return a zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome

Possible Return Values

(0) Service Discovery Start success

(-4) Function_Error

API Call

GDIS_Service_Discovery_Start(BluetoothStackID, ConnectionID, (sizeof(UUID)/sizeof(GATT_UUID_t)), UUID, GDIS_Event_Callback, sdHRS);

API Prototype

int BTPSAPI GDIS_Service_Discovery_Start(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GDIS_Event_Callback_t ServiceDiscoveryCallback, unsigned long ServiceDiscoveryCallbackParameter)

Description of API

The GDIS_Service_Discover_Start is in an application module called GDIS that is provided to allow an easy way to perform GATT service discovery. This module can and should be modified for the customers use. This function is called to start a service discovery operation by the GDIS module.

ConfigureRemoteHRS

Description

The ConfigureRemoteHRS command is responsible for configuring a HRP Service on a remote device. This command will return zero on successful execution and a negative value on all errors.

Parameters

The ConfigureRemoteHRS requires only one parameter, Heart Rate Notify. 0 disables Heart Rate Notify and 1 enables Heart Rate Notify

Command Call Examples

ConfigureRemoteHRS 1 configures the service with Heart Rate Notify enabled.

ConfigureRemoteHRS 0 configures the service with Heart Rate Notify disabled.

Possible Return Values

(0) Successfully configured HRS on the remote device

(-4) FUNCTION_ERROR

NotifyHeartRate

Description

The NotifyHeartRate command is responsible for performing a Heart Rate Measurement notification to a connected remote device. This command will return zero on successful execution and a negative value on all errors.

Parameters

the NotifyHeartRate command requires 5 parameters. The first parameter is the BPM (beats per minute). The second is the Heart Rate format, 0 = Byte, 1 = Word. The third is the Sensor Contact Status, 0 = Not Supported, 1 = Supported/Not Detected, 2 = Supported/Detected. The fourth is Energy Expended, 0 = Don't Send, 1 = Send. The fifth and last is the number of RR Intervals, 0 = None, X = Number of Intervals.

Command Call Examples

NotifyHeartRate 80 1 2 1 0

Possible Return Values

(0) Successfully sent the Heart Rate Measurement

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) HRS_ERROR_INVALID_PARAMETER

(-1002) HRS_ERROR_INSUFFICIENT_RESOURCES

(-1004) HRS_ERROR_INVALID_INSTANCE_ID

API Call

HRS_Notify_Heart_Rate_Measurement(BluetoothStackID, HRSInstanceID, ConnectionID, HeartRatePtr)

API Prototype

Int BTPSAPI HRS_Notify_Heart_Rate_Measurement(unsigned int BluetoothStackID, unsigned int InstanceID, unsigned int ConnectionID, HRS_Heart_Rate_Measurement_Data_t *Heart_Rate_Measurement)

Description of API

This function is responsible for sending a Heart Rate Measurement to a specified remote device. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to HRS_Initialize_Server(). The third parameter is the ConnectionID of the remote device to send the notification to. The final parameter is the Heart Rate Measurement data to notify. This function will return zero if the notification was sent successfully or a negative return error code if there was an error condition.

GetBodySensorLocation

Description

The GetBodySensorLocation command is responsible for reading the Body Sensor Location characteristic. It can be executed by a Server or a Client with an open connection to a remote Server. This command will return zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome

Possible Return Values

(0) Successfully Set Body Sensor Location

(-4) FUNCTION_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000) HRS_ERROR_INVALID_PARAMETER

(-1004) HRS_ERROR_INVALID_INSTANCE_ID

API Call

HRS_Query_Body_Sensor_Location(BluetoothStackID, HRSInstanceID, &location)

API Prototype

int BTPSAPI HRS_Query_Body_Sensor_Location(unsigned int BluetoothStackID, unsigned int InstanceID, Byte_t *Body_Sensor_Location)

Description of API

This function is responsible for querying the current location of the body sensor for the specified HRS instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to HRS_Initialize_Server(). The final parameter is a pointer to return the current Body Sensor Location for the specified HRS instance. This function will return zero if the body sensor location was successfully set or a negative return error code if there was an error condition.

SetBodySensorLocation

Description

The SetBodySensorlocation command is responsible for writing the Body Sensor Location characteristic. It can be executed only by a Server. This command will return zero on successful execution and a negative value on all errors.

Parameters

The SetBodySensorLocation command requires one parameter which is the Location of the Heart Rate sensor. This parameter should be an integer between 1 and 6 with each number corresponding to a different location.

0 – Other, 1 – Chest, 2 – Wrist, 3 – Finger, 4 – Hand, 5 - Ear Lobe, 6 – Foot.

Command Call Examples

"SetBodySensorLocation 3" sets the sensor location to Finger.

"SetBodySensorLocation 5" sets the sensor location to Ear lobe.

Possible Return Values

(0) Successfully Set Body Sensor Location

(-4) FUNCTION_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000) HRS_ERROR_INVALID_PARAMETER

(-1004) HRS_ERROR_INVALID_INSTANCE_ID

API Call

HRS_Set_Body_Sensor_Location(BluetoothStackID, HRSInstanceID, (Byte_t)TempParam->Params[0].intParam

API Prototype

Int BTPSAPI HRS_Set_Body_Sensor_Location(unsigned int BluetoothStackID, unsigned int InstanceID, Byte_t Body_Sensor_Location)

Description of API

This function is responsible for setting the location of the body sensor for the specified HRS instance. The Body_Sensor_Location parameter should be an enumerated value of the form HRS_BODY_SENSOR_LOCATION_XXX. This function will return zero if the body sensor location was successfully set or a negative return error code if there was an error condition.

ResetEnergyExpended

Description

The ResetEnergyExpended command is responsible for writing the Reset Energy Expended command to a remote Server Control Point. It can be executed only by a Client. This command will return a zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) successfully reset energy expended (-1000) HRS_ERROR_INVALID_PARAMETER

API Call

HRS_Format_Heart_Rate_Control_Command(ccResetEnergyExpended, HRS_HEART_RATE_CONTROL_POINT_VALUE_LENGTH, CommandBuffer)

API Prototype

int BTPSAPI HRS_Format_Heart_Rate_Control_Command(HRS_Heart_Rate_Control_Command_t Command, unsigned int BufferLength, Byte_t *Buffer)

Description of API

The Format_Heart_Rate_Control_Command function is responsible for formatting a Heart Rate Control Command into a user specified buffer. The first parameter is the command to format. The final two parameters contain the length of the buffer, and the buffer, to format the command into. This function returns a zero if successful or a negative return error code if an error occurs. The BufferLength and Buffer parameter must point to a buffer of at least HRS_HEART_RATE_CONTROL_POINT_VALUE_LENGTH in size. HRS_Heart_Rate_Control_Command_t is a ccResetEnergyExpended command which is placed in the buffer.

16 HTP Demo Guide

16.1 Demo Overview

Note

The same instructions can also be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms.

The Health Thermometer profile enables a Collector device to connect and interact with a Thermometer sensor for use in healthcare applications. There are two roles defined in the profile. The first is the thermometer which measures the temperature and the other is collector which recieves the temperature and settings from the Thermometer. Typically the Thermometer would be present physically on the person/object whose temperature is measured at different locations like the mouth or the armpit. The collector device would be present somewhere close by getting regular temperature updates at specified intervals from the remote thermometer.

This application allows the user to use a console to use Bluetooth Low Energy (BLE) to establish connection between two BLE devices, notify temperature, change measurement intervals and get and change the location of the thermometer sensor.

It is recommended that the user visits the kit setup Getting Started Guide for MSP430, Getting Started Guide for TIVA, Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as MSP-EXP430F5438 USB - Serial Port (COM x), Tiva Virtual COM Port (COM x), XDS110 Class Application/User UART (COM x) for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud (9600 for MSP430), 8, n, 1. Once connected, reset the device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

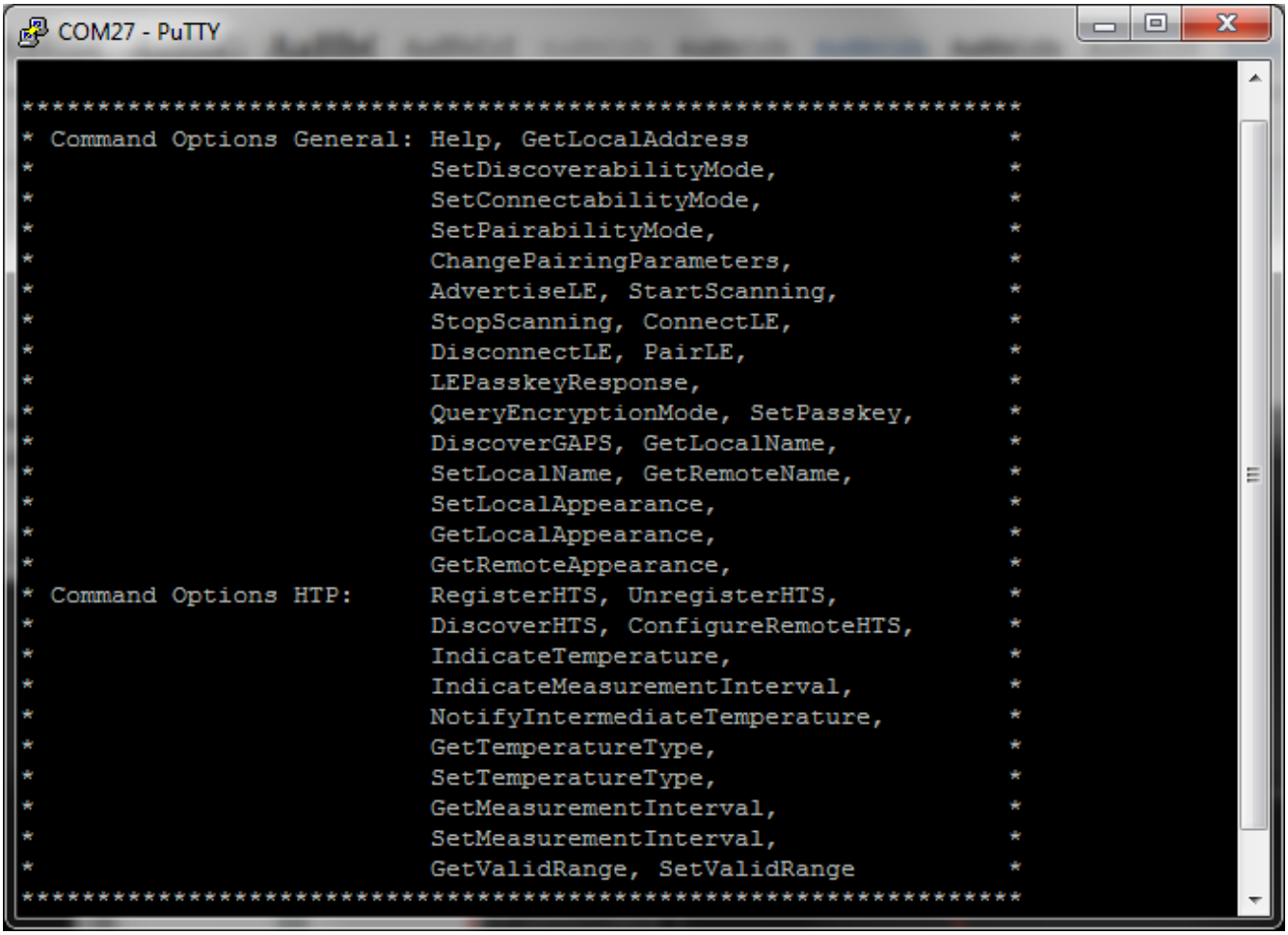

Figure 16-1. HTP Demo Start

16.2 Demo Application

The demo application provides a description on how to use the demo application to connect two configured boards and communicate over Bluetooth Low Energy (BLE). The included application registers a custom service on a board when the stack is initialized.

Device 1 (Server) setup on the demo application

1. To start with one of the devices has to have the Phone Alert Status running on it. It can be started by issuing the RegisterHTP command.
2. Next, the device acting as a Server needs to advertise to other devices. This can be done by issuing the AdvertiseLE 1.

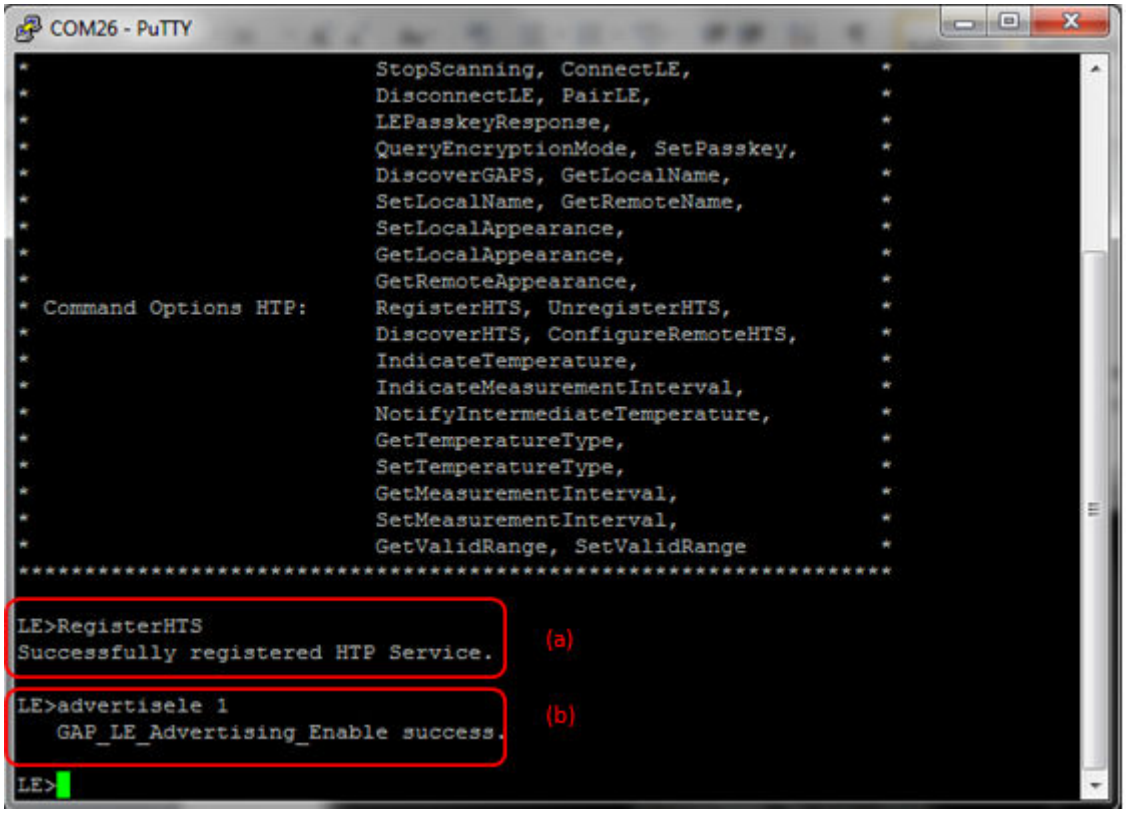

Figure 16-2. HTP Demo Register HTS

Device 2 (Client) setup on the demo application

Note

Steps c and d are optional if you already know the Bluetooth address of the device that you want to connect to.

1. The Client LE device can try to find which LE devices are in the vicinity issuing the StartScanning command.
2. Once you have found the device, you can stop scanning by issuing the StopScanning command.

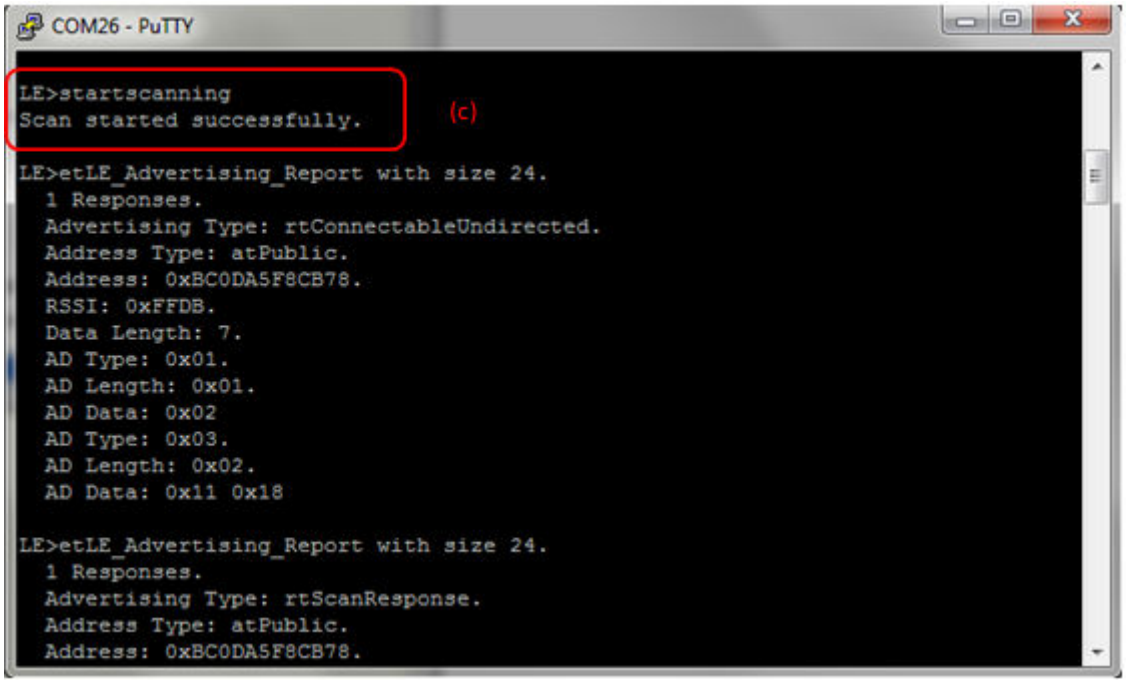

Figure 16-3. HTP Demo scanning

Initiating connection from device 2

1. Once the application on the Client side knows the Bluetooth address of the device that is advertising, it can connect to that device issuing the ConnectLE <Bluetooth Address> command.

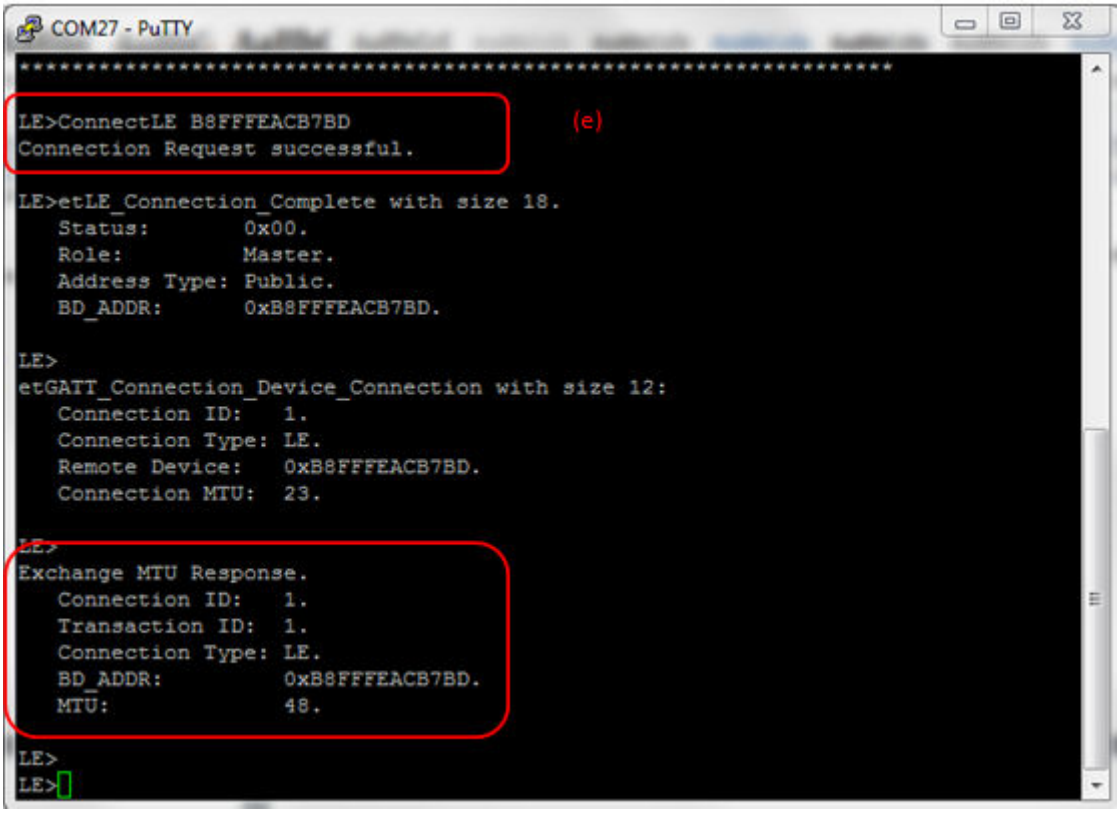

Figure 16-4. HTP Demo Connect

Identify supported services

1. After initialization, the Client needs to find out whether HTS services are supported and what HTS features are available. For this, the DiscoverHTS command is run on the Client. After the service discovery operation is complete, the HTS Service Discovery Summary is shown and list of supported features is shown.

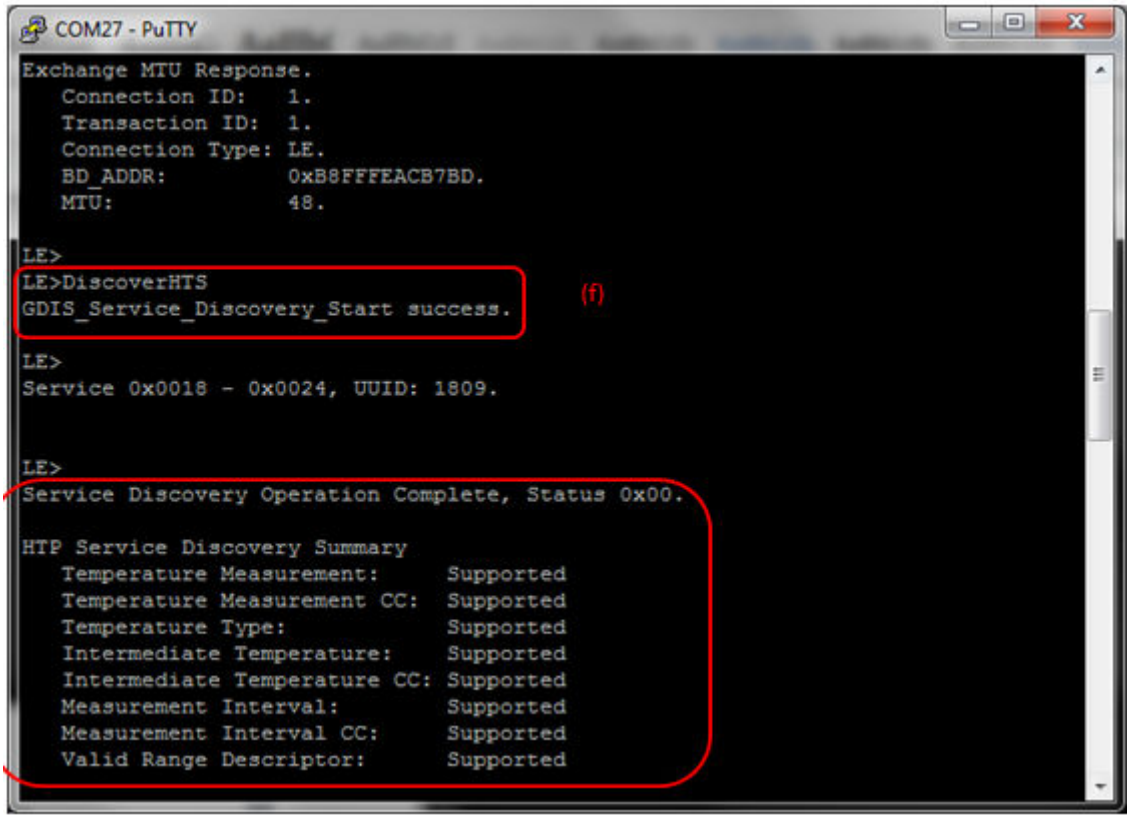

Figure 16-5. HTP Demo Discover HTS

2. After getting the list of supported features, the next step is to configure HTS on the Client. We can enable support for various notifications by issuing the ConfigureRemoteHTS <Temperature Measurement Indicate> <measurement Interval Indicate> <Intermediate Termature Notify> command.

```
LE>ConfigureRemoteHTS
Usage: ConfigureRemoteHTS [Temperature Measurment Indicate (0 = disable, 1 = enable)]
                          [Measurement Interval Indicate (0 = disable, 1 = enable)]
                          [Intermediate Temperature Notify (0 = disable, 1 = enable)]
```

Figure 16-6. HTP Demo Configure Remote HTS

3. In our case we enable all 3.

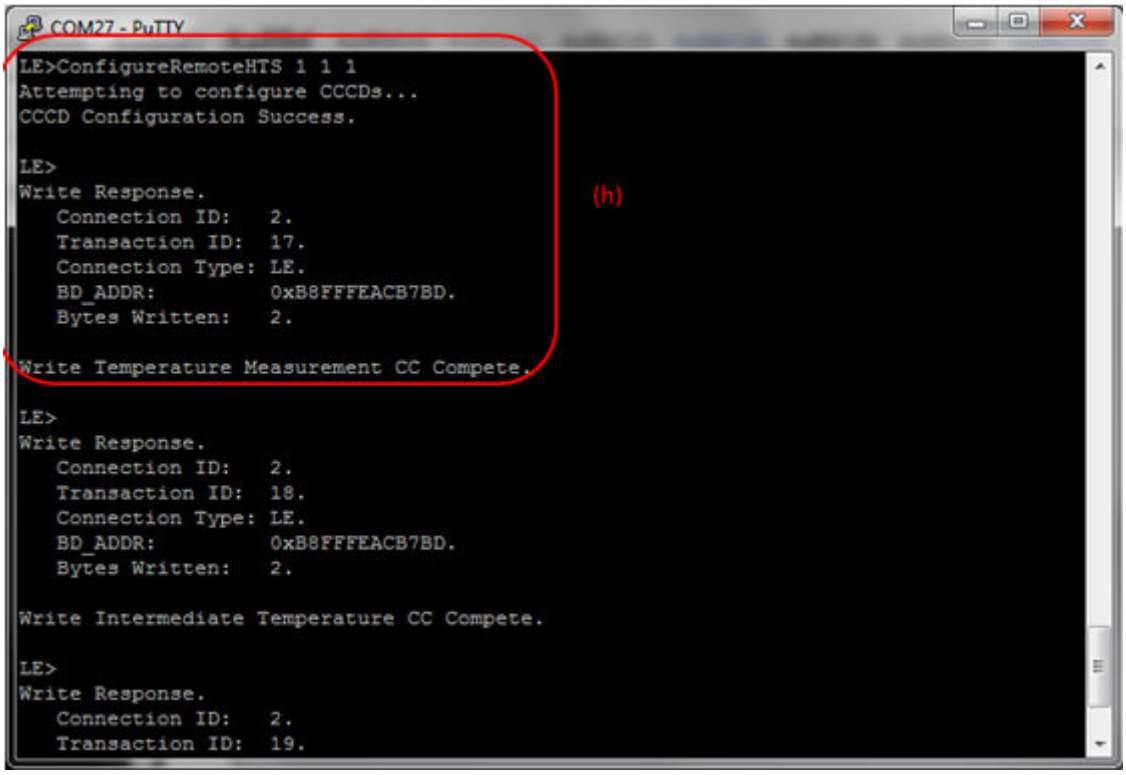

Figure 16-7. HTP Demo Configure Remote HTS Results

Temperature Measurement Indication

1. After configuring, the Health Thermometer Service is now active. We can indicate the temperature with by issuing the IndicateTemperature <Exponent> <mantissa> <Units> <Temperature Type> command.

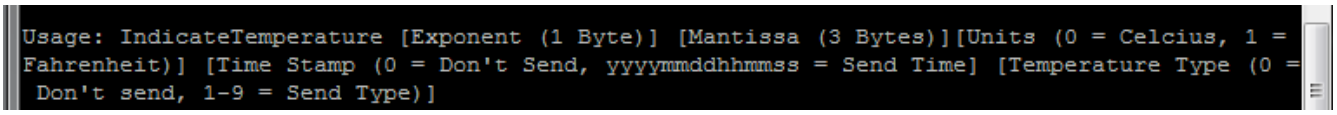

Figure 16-8. HTP Demo Indicate Temperature

Note

To set a temperature of 100 F with a timestamp of 7/24/2012 9:50AM at the body we would give the command as IndicateTemperature 0 100 1 201207240950002

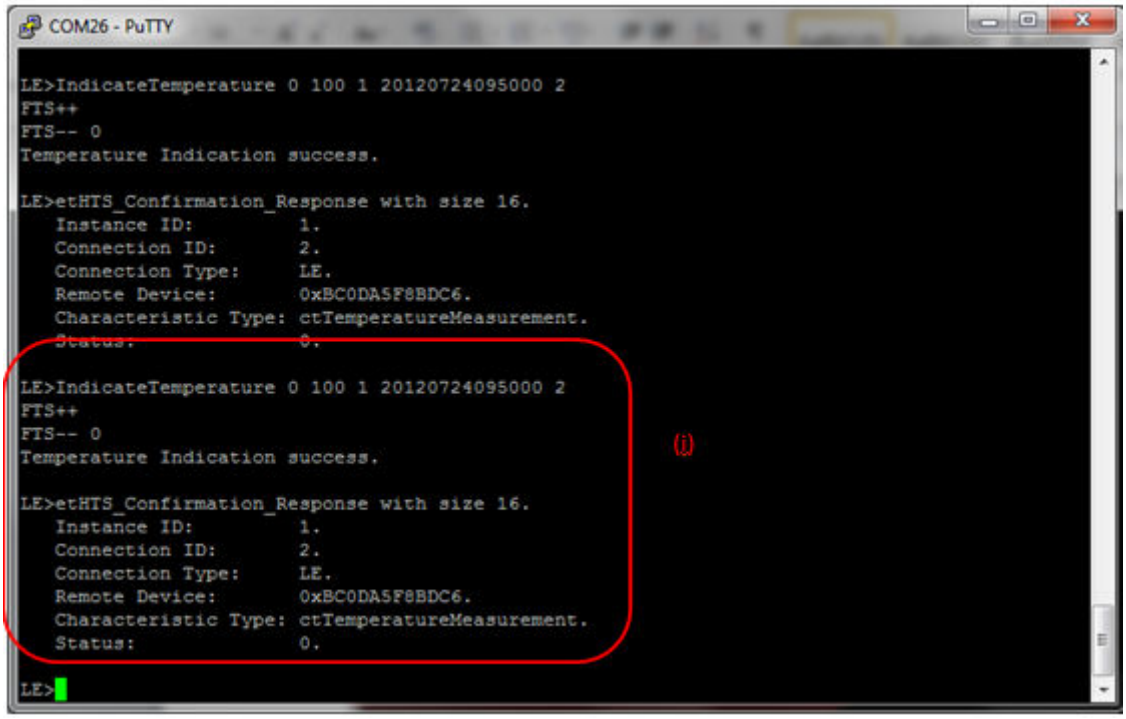

Figure 16-9. HTP Demo Indicate Temperature Result

The Client gets the temperature as follows:

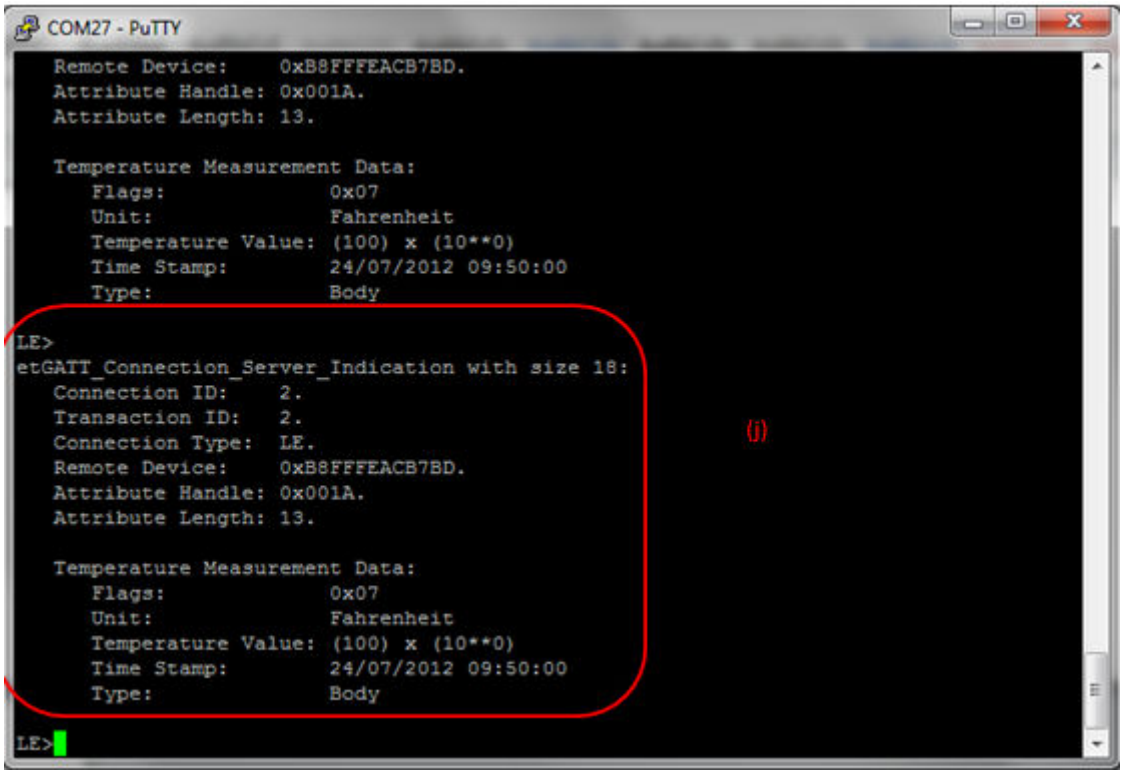

Figure 16-10. HTP Demo Temperature Indication

16.3 Application Commands

RegisterHTS

Description

The RegisterHTS command is responsible for registering a HTP Service. This command will return zero on successful execution and a negative value on all errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully registered an HTP Server

(-4) FUNCTION_ERROR

(-1000) HTS_ERROR_INVALID_PARAMETER

(-1002) HTS_ERROR_INSUFFICIENT_RESOURCES

(-1003) HTS_ERROR_SERVICE_ALREADY_REGISTERED

API Call

HTS_Initialize_Service(BluetoothStackID, HTS_EventCallback, NULL, &HTSInstanceID);

API Prototype

int BTPSAPI HTS_Initialize_Service(unsigned int BluetoothStackID, HTS_Event_Callback_t EventCallback, unsigned long CallbackParameter, unsigned int *ServiceID)

Description of API

This function is responsible for opening a HTS Server. The first parameter is the Bluetooth Stack ID on which to open the Server. The second parameter is the Callback function to call when an event occurs on this Server Port. The third parameter is a user-defined callback parameter that will be passed to the callback function with each event. The final parameter is a pointer to store the GATT Service ID of the registered HTS service. This can be used to include the service registered by this call. This function returns the positive, non-zero, Instance ID or a negative error code.

UnRegisterHTS

Description

The UnRegisterHTS command is responsible for unregistering a HTP Service. This command will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Successfully unregistered the HTP Server

(-4) FUNCTION_ERROR

(-1000) HTS_ERROR_INVALID_PARAMETER

(-1004) HTS_ERROR_INVALID_INSTANCE_ID

API Call

HTS_Cleanup_Service(BluetoothStackID, HTSInstanceID);

API Prototype

int BTPSAPI HTS_Cleanup_Service(unsigned int BluetoothStackID, unsigned int InstanceID)

Description of API

This function is responsible for closing a previously opened HTS Server. The first parameter is the Bluetooth Stack ID on which to close the Server. The second parameter is the InstanceID that was returned from a

successfull call to HTS_Initialize_Service(). This function returns a zero if successful or a negative return error code if an error occurs.

DiscoverHTS

Description

The DiscoverHTS command is responsible for performing a HTP Service Discovery Operation. This command will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully performed HTP Service Discovery Operation

(-4) FUNCTION_ERROR

API Call

GDIS_Service_Discovery_Start(BluetoothStackID, ConnectionID, (sizeof(UUID)/sizeof(GATT_UUID_t)), UUID, GDIS_Event_Callback, sdHTS);

API Prototype

int BTPSAPI GDIS_Service_Discovery_Start(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GDIS_Event_Callback_t ServiceDiscoveryCallback, unsigned long ServiceDiscoveryCallbackParameter)

Description of API

The GDIS_Service_Discover_Start is in an application module called GDIS that is provided to allow an easy way to perform GATT service discovery. This module can and should be modified for the customers use. This function is called to start a service discovery operation by the GDIS module.

ConfigureRemoteHTS

Description

The ConfigureRemoteHTS command is responsible for configuring a HTP Service on a remote device. This command will return zero on successful execution and a negative value on errors.

Parameters

This function required three parameters. The first is Temperature Measurement Indicate, 0 = disable, 1 = enable. The second is Measurement Interval Indicate, 0 = disable, 1 = enable. The third is Intermediate Temperature Notify, 0 = disable, 1 = enable.

Command Call Examples

"ConfigureRemoteHTS 1 1 1" enables support for all three; Temperature Measurement Indicate, Measurement Interval Indicate, and Intermediate Temperature Notify.

Possible Return Values

(0) Successfully configured the HTP Service.

(-4) FUNCTION_ERROR

IndicateTemperature

Description

The IndicateTemperature command is responsible for sending a temperature measurement indication to a connected remote device. This command returns zero on successful execution and a negative value on errors.

Parameters

This function requires five parameters. The first is exponent (1 byte). The second is Mantissa (3 bytes). The third is units, 0 = Celcius, 1 = Fahrenheit. The fourth is the time stamp in the form yyyymmddhhmmss which is the send time in year-month-day-hour-minute-second, 0 = don't send. The fifth is the temperature type, 0 = dot send, 1-9 = send type.

Command Call Examples

"IndicateTemperature 0 100 1 20120724095000 2" sets a temperature of 100 F with a timestamp of 7/24/2012 9:50AM.

Possible Return Values

(0) Successfully sent Temperature Indication

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR (-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000) HTS_ERROR_INVALID_PARAMETER

(-1002) HTS_ERROR_INSUFFICIENT_RESOURCES

(-1004) HTS_ERROR_INVALID_INSTANCE_ID

API Call

HTS_Indicate_Temperature_Measurement(BluetoothStackID, HTSInstanceID, ConnectionID, &TemperatureMeasurement)

API Prototype

int BTPSAPI HTS_Indicate_Temperature_Measurement(unsigned int BluetoothStackID, unsigned int InstanceID, unsigned int ConnectionID, HTS_Temperature_Measurement_Data_t *Temperature_Measurement)

Description of API

This function is responsible for sending a Temperature Measurement indication to a specified remote device. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to HTS_Initialize_Server(). The third parameter is the ConnectionID of the remote device to send the indication to. The final parameter is the Temperature Measurement data to indicate. This function returns a zero if successful or a negative return error code if an error occurs.

IndicateMeasurementInterval

Description

The IndicateMeasurementInterval command is responsible for sending a measurement interval indication to a connected remote device. This command will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully sent the measurement interval indication.

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000) HTS_ERROR_INVALID_PARAMETER

(-1004) HTS_ERROR_INVALID_INSTANCE_ID

(-1006) HTS_ERROR_INDICATION_OUTSTANDING

API Call

HTS_Indicate_Measurement_Interval(BluetoothStackID, HTSInstanceID, ConnectionID)

API Prototype

int BTPSAPI HTS_Indicate_Measurement_Interval(unsigned int BluetoothStackID, unsigned int InstanceID, unsigned int ConnectionID)

Description of API

This function is responsible for sending a Measurement Interval indication to a specified remote device. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to HTS_Initialize_Server(). The third parameter is the ConnectionID of the remote device to send the notification to. This function returns a zero if successful or a negative return error code if an error occurs.

NotifyIntermediateTemperature

Description

the NotifyIntermediateTemperature command is responsible for sending an intermediate temperature notification to a connected remote device. This command will return zero on successful execution and a negative value on errors.

Parameters

This command requires five parameters. The first is the exponent (1 byte). The second is Mantissa (the temperature) (3 bytes). The third is units, 0 = Celcius, 1 = Fahrenheit. The fourth is the time stamp in the form yyyymmddhhmmss which is the send time in year-month-day-hour-minute-second, 0 = don't send. The fifth is the temperature type, 0 = don't send, 1-9 = send type.

Command Call Examples

"NotifyIntermediateTemperature 0 100 1 20120724095000 2" sends a temperature of 100 F with a timestamp of 7/24/2012 9:50AM.

Possible Return Values

(0) Successfully sent intermediate temperature notification

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE (-1000) HTS_ERROR_INVALID_PARAMETER

(-1002) HTS_ERROR_INSUFFICIENT_RESOURCES (-1004) HTS_ERROR_INVALID_INSTANCE_ID

API Call

HTS_Notify_Intermediate_Temperature(BluetoothStackID, HTSInstanceID, ConnectionID, &TemperatureMeasurement)

API Prototype

int BTPSAPI HTS_Notify_Intermediate_Temperature(unsigned int BluetoothStackID, unsigned int InstanceID, unsigned int ConnectionID, HTS_Temperature_Measurement_Data_t *Temperature_Measurement)

Description of API

This function is responsible for sending an Intermediate Temperature notification to a specified remote device. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to HTS_Initialize_Server(). The third parameter is the ConnectionID of the remote device to send the notification to. The final parameter is the Intermediate Temperature data to notify. This function returns a zero if successful or a negative return error code if an error occurs.

GetTemperatureType

Description

The GetTemperatureType command is responsible for reading the temperature type. It can be executed by a Server or a Client with an open connection to a remote Server. If executed as a Client, a GATT read request will be generated, and the results will be returned as a response in the GATT Client event callback. This command will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully queried the temperature type

(-4) FUNCTION_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000) HTS_ERROR_INVALID_PARAMETER

(-1004) HTS_ERROR_INVALID_INSTANCE_ID

API Call

HTS_Query_Temperature_Type(BluetoothStackID, HTSInstanceID, &tempType)

API Prototype

int BTPSAPI HTS_Query_Temperature_Type(unsigned int BluetoothStackID, unsigned int InstanceID, Byte_t *Temperature_Type)

Description of API

This function is responsible for querying the current Temperature Type on the specified HTS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to HTS_Initialize_Server(). The final parameter is a pointer to return the current Temperature Type for the specified HTS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

SetTemperatureType

Description

The SetTemperatureType command is responsible for writing the temperature type. It can be executed only by a Server. This command will return zero upon successful execution and a negative value on errors.

Parameters

This Command requires one parameter which is the temperature type (1-9).

HTS_TEMPERATURE_TYPE_ARMPIT 1

HTS_TEMPERATURE_TYPE_BODY 2

HTS_TEMPERATURE_TYPE_EAR 3

HTS_TEMPERATURE_TYPE_FINGER 4

HTS_TEMPERATURE_TYPE_GASTRO_INTESTINAL_TRACT 5

HTS_TEMPERATURE_TYPE_MOUTH 6

HTS_TEMPERATURE_TYPE_RECTUM 7

HTS_TEMPERATURE_TYPE_TOE 8

HTS_TEMPERATURE_TYPE_TYMPANUM 9

Command Call Examples

"SetTemperatureType 3" Sets the Temperature Type to EAR.

"SetTemperatureType 8" Sets the Temperature Type to TOE.

Possible Return Values

(0) Successfully set the Temperature Type.

(-4) FUNCTION_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000) HTS_ERROR_INVALID_PARAMETER

(-1004) HTS_ERROR_INVALID_INSTANCE_ID

API Call

HTS_Set_Temperature_Type(BluetoothStackID, HTSInstanceID, (Byte_t)TempParam->Params[0].intParam)

API Prototype

int BTPSAPI HTS_Set_Temperature_Type(unsigned int BluetoothStackID, unsigned int InstanceID, Byte_t Temperature_Type)

Description of API

This function is responsible for setting the Temperature Type on the specified HTS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to HTS_Initialize_Server(). The final parameter is the Temperature Type to set for the specified HTS Instance. This function returns a zero if successful or a negative return error code if an error occurs. The Temperature_Type parameter should be an enumerated value of the form HTS_TEMPERATURE_TYPE_XXX.

GetMeasurementInterval

Description

The GetMeasurementInterval command is responsible for reading the measurement interval. It can be executed by a Server or a Client with an open connection to a remote Server. If executed as a Client, a GATT read request will be generated, and the results will be returned as a response in the GATT Client event callback. This command will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully read the measurement interval

(-4) FUNCTION_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE (-1000) HTS_ERROR_INVALID_PARAMETER (-1004) HTS_ERROR_INVALID_INSTANCE_ID

API Call

HTS_Query_Measmureent_Interval(BluetoothStackID, HTSInstanceID, &Interval)

API Prototype

int BTPSAPI HTS_Query_Measurement_Interval(unsigned int BluetoothStackID, unsigned int InstanceID, Word_t *Measurement_Interval)

Description of API

This function is responsible for querying the current Measurement Interval on the specified HTS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to HTS_Initialize_Server(). The final parameter is a pointer to return the current Measurement Interval for the specified HTS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

SetMeasurementInterval

Description

The SetMeasurementInterval command is responsible for writing the measurement interval. It can be executed by a Server or as a Client with an open connection to a remote Server if Client write access is supported. If executed as a Client, a GATT write request will be generated, and the results will be returned as a response in the GATT Client event callback. This command will return zero on successful execution and a negative value on errors.

Parameters

This command requires one parameter which is the Measurement Interval.

Command Call Examples

"SetMeasurementInterval 2000" sets the Measurement Interval to 2000

Possible Return Values

(0) Successfully set the Measurement Interval

(-4) FUNCTION_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000) HTS_ERROR_INVALID_PARAMETER

(-1004) HTS_ERROR_INVALID_INSTANCE_ID

API Call

HTS_Set_Measurement_Interval(BluetoothStackID, HTSInstanceID, (Word_t)TempParam->Params[0].intParam)

API Prototype

int BTPSAPI HTS_Set_Measurement_Interval(unsigned int BluetoothStackID, unsigned int InstanceID, Word_t Measurement_Interval)

Description of API

This function is responsible for setting the Measurement Interval on the specified HTS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to HTS_Initialize_Server(). The final parameter is the Measurement Interval to set for the specified HTS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

GetValidRange

Description

The GetValidRange command is responsible for reading the valid range. It can be executed by a Server or as a Client with an open connection to a remote Server if Client write access to the measurement interval is supported. If executed as a Client, a GATT read request will be generated, and the results will be returned as a response in the GATT Client event callback. This command will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully read the range

(-4) FUNCTION_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000) HTS_ERROR_INVALID_PARAMETER

(-1004) HTS_ERROR_INVALID_INSTANCE_ID

API Call

HTS_Query_Valid_Range(BluetoothStackID, HTSInstanceID, &ValidRange)

API Prototype

int BTPSAPI HTS_Query_Valid_Range(unsigned int BluetoothStackID, unsigned int InstanceID, HTS_Valid_Range_Data_t *ValidRange)

Description of API

This function is responsible for querying the Valid Range descriptor value on the specified HTS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to HTS_Initialize_Server(). The final parameter is a pointer to store the Valid Range structure for the specified HTS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

SetValidRange

Description

The SetValidRange command is responsible for writing the valid range. It can be executed only by a Server. This command will return zero on successful execution and a negative value on errors.

Parameters

This command requires two parameters. The first is the lower bound of the range and the second is the upper bound.

Possible Return Values

(0) Successfully set the range

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-103) BTPS_ERROR_FEATURE_NOT_AVAILABLE

(-1000) HTS_ERROR_INVALID_PARAMETER

(-1004) HTS_ERROR_INVALID_INSTANCE_ID

API Call

HTS_Set_Valid_Range(BluetoothStackID, HTSInstanceID, &ValidRange)

API Prototype

int BTPSAPI HTS_Set_Valid_Range(unsigned int BluetoothStackID, unsigned int InstanceID, HTS_Valid_Range_Data_t *ValidRange)

Description of API

This function is responsible for setting the Valid Range descriptor value on the specified HTS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to HTS_Initialize_Server(). The final parameter is the Valid Range to set for the specified HTS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

17 PASP Demo Guide

17.1 Demo Overview

Note

The same instructions can be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms. The same instructions can be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms.

The phone alert state profile (PASP) is used to notify the Client about the Alert and Ringer Status of the Server device. There are two roles defined in this profile, Server and the Client. Any changes made to the alert or Ringer status is also notified by the Server to the Client. The Client may also request the Server to put the phone into Silent mode and bring it out of Silent mode.

This application allows the user to use a console to use Bluetooth Low Energy (BLE) to establish connection between two BLE devices, change phone alert status, change ringer status, put in and get out of silent mode.

It is recommended that the user visits the kit setup Getting Started Guide for MSP430, Getting Started Guide for TIVA, Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as MSP-EXP430F5438 USB - Serial Port (COM x), Tiva Virtual COM Port (COM x), XDS110 Class Application/User UART (COM x) for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud (9600 for MSP430), 8, n, 1. Once connected, reset the device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

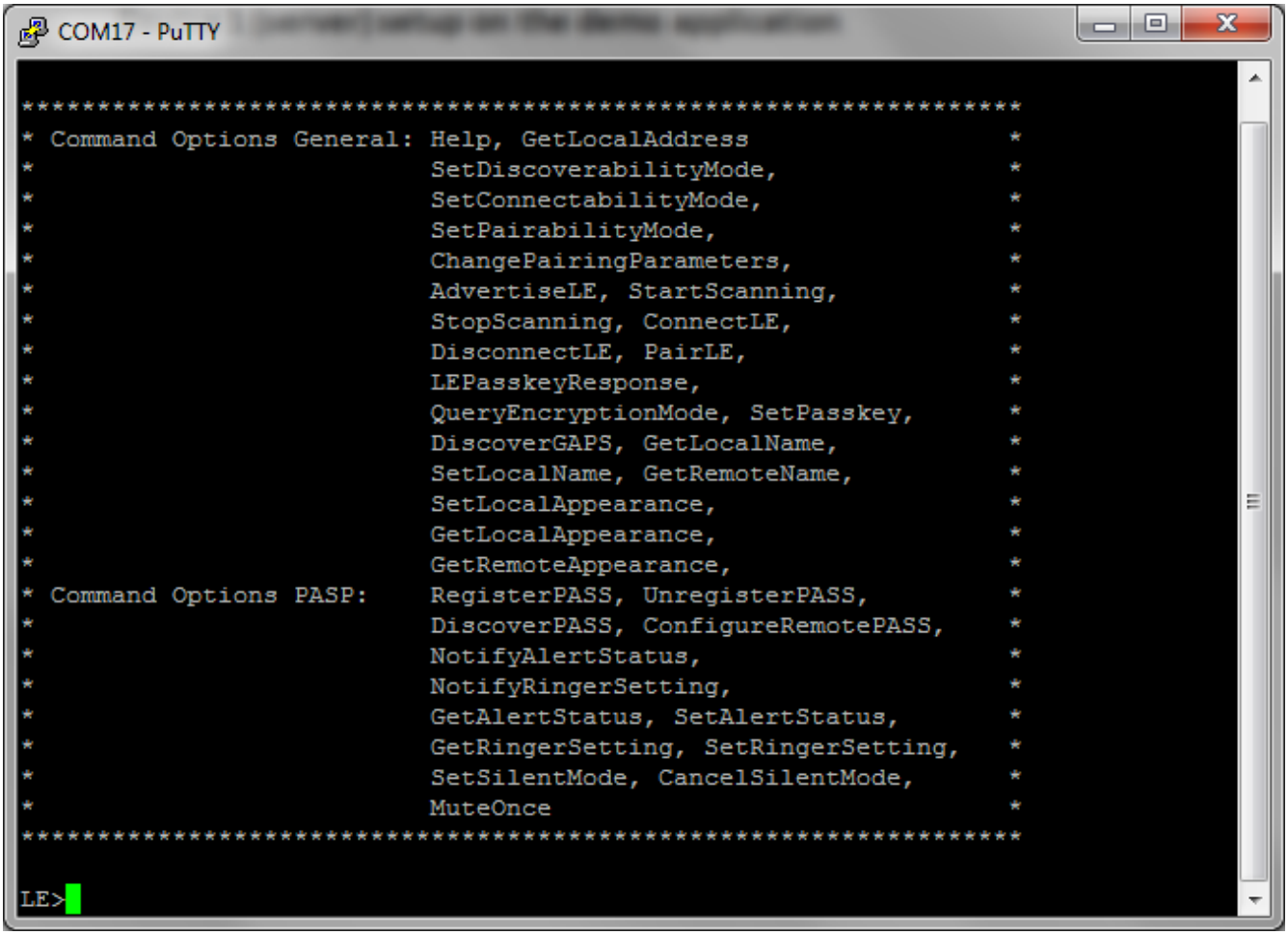

Figure 17-1. PASP Demo Start Terminal

17.2 Demo Application

The demo application provides a description on how to use the demo application to connect two configured boards and communicate over Bluetooth LE. The included application registers a custom service on a board when the stack is initialized.

Device 1 (Server) setup on the demo application

1. To start with one of the devices has to have the Phone Alert Status running on it. It can be started by running RegisterPASS.
2. Next, the device acting as a Server needs to advertise to other devices. This can be done by issuing the AdvertiseLE 1 command.

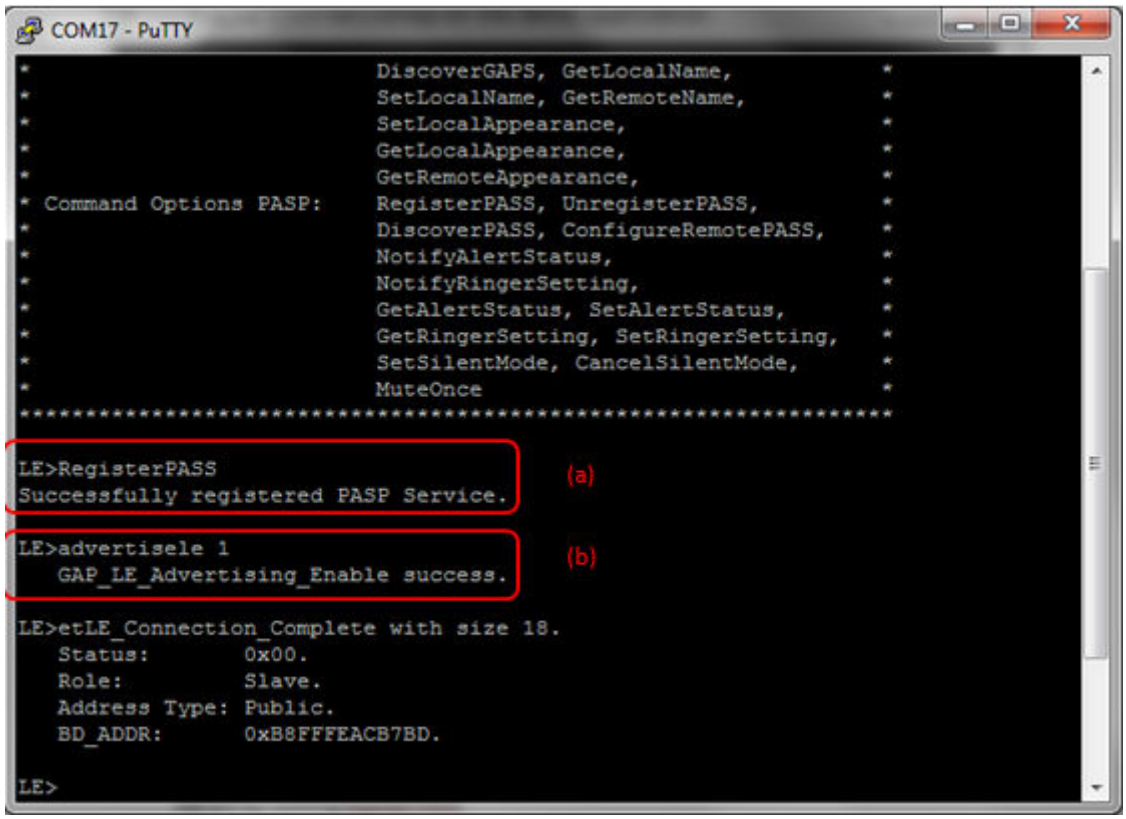

Figure 17-2. PASP Demo Register PASS

Device 2 (Client) setup on the demo application

Note

Steps c and d are optional if you already know the Bluetooth address of the device that you want to connect to.

1. The Client LE device can try to find which LE devices are in the vicinity issuing the StartScanning command.
2. Once you have found the device, you can stop scanning by issuing the StopScanning command.

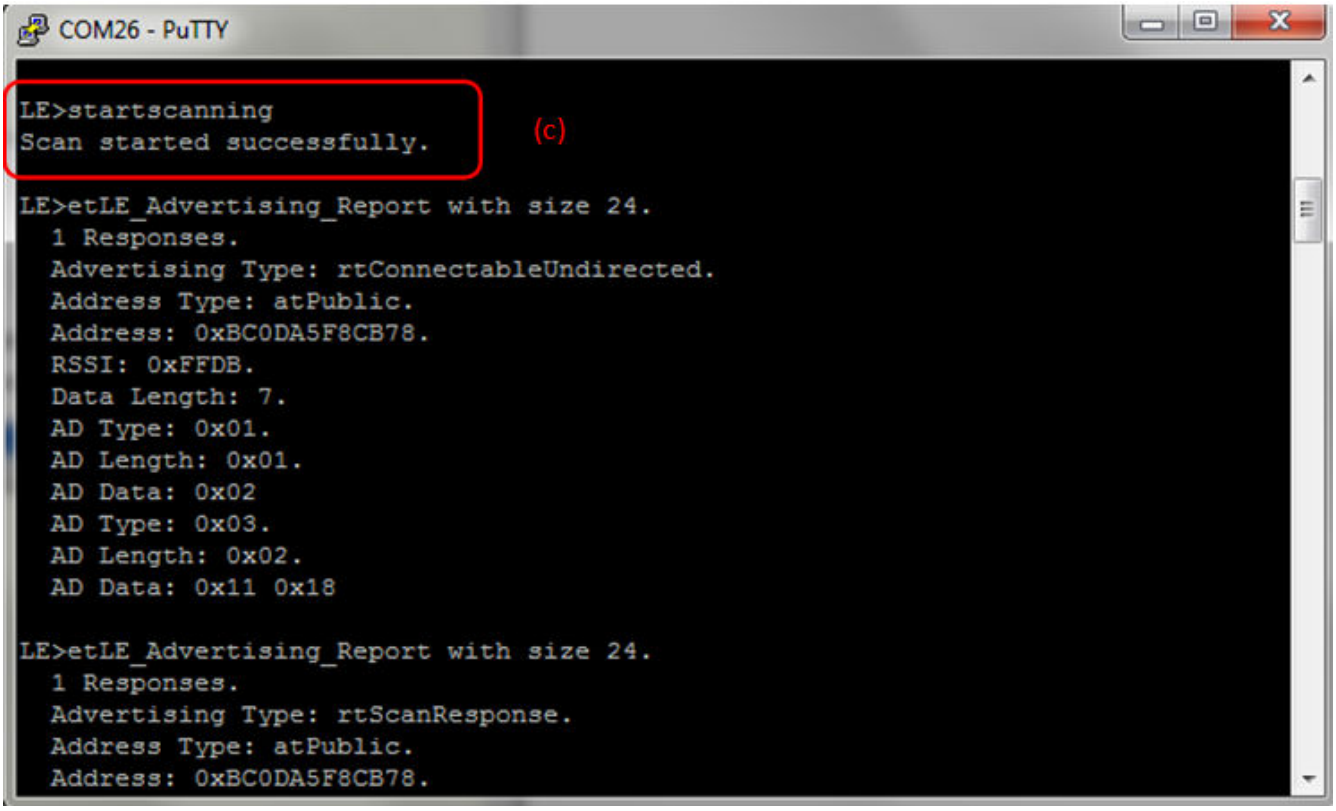

Figure 17-3. PASP Demo Start Scanning

Initiating connection from device 2

1. Once the application on the Client side knows the Bluetooth address of the device that is advertising, it can connect to that device issuing the ConnectLE <Bluetooth Address> command

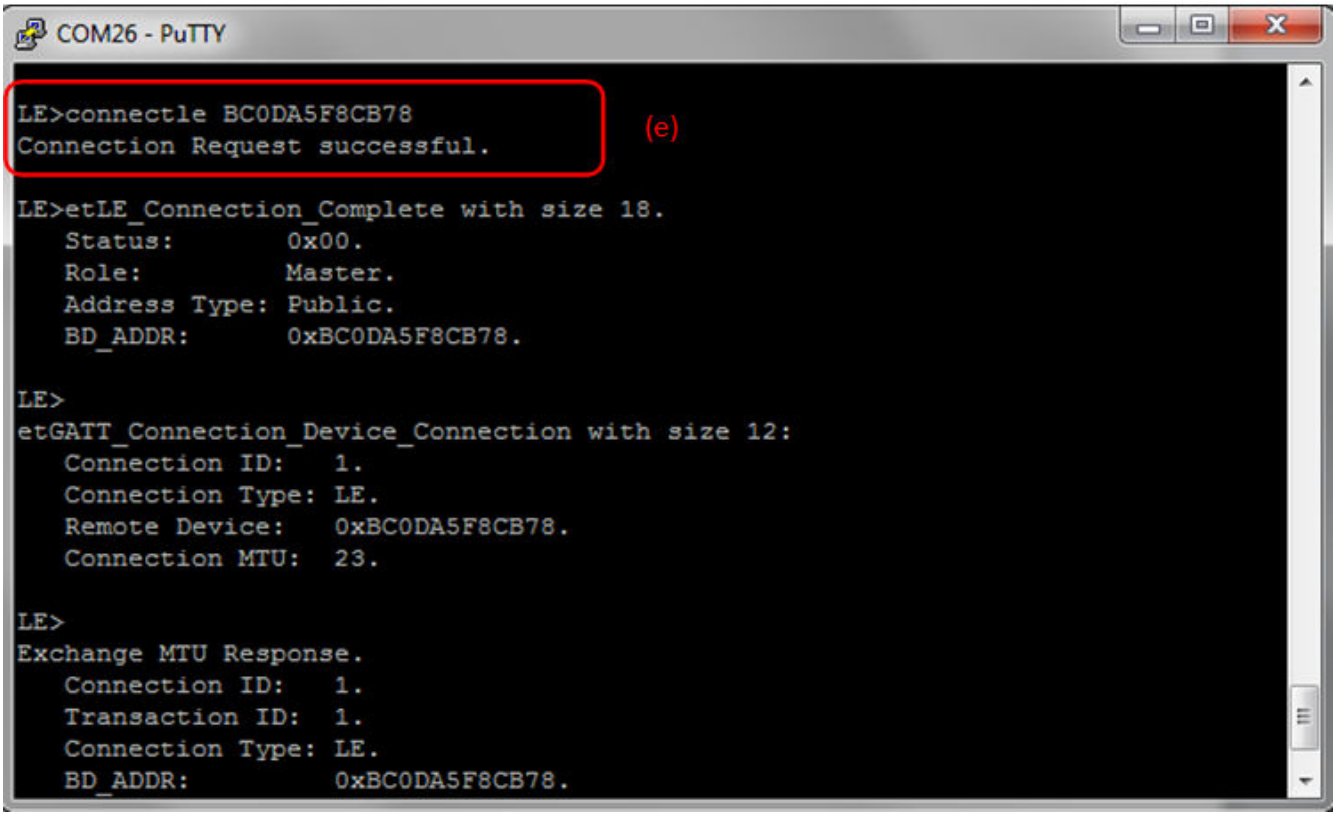

Figure 17-4. PASP Connection Request Successful

2. After initialization, the Client needs to find out whether PASS services are supported and what PASS features are available. For this, issue the DiscoverPASS command on the Client. After the service discovery operation is complete, the PASS Service Discovery Summary and list of supported features is shown.

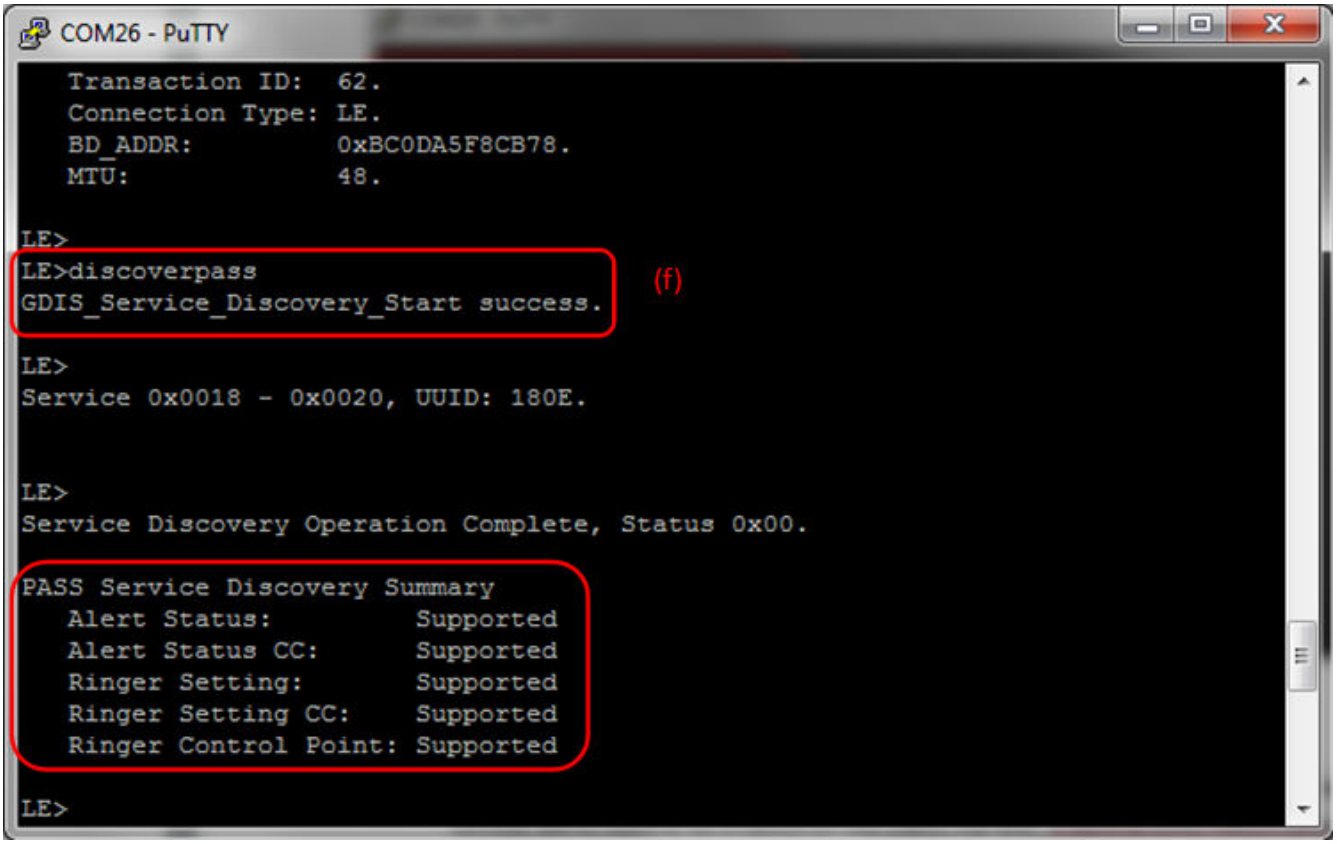

Figure 17-5. PASP Demo Discover PASS

3. After getting the list of supported features, the next step is to configure the PASS on the Client. We can set the configuration for Alert Status Notify and Ringer Settings.

```
LE>ConfigureRemotePASS
Usage: ConfigureRemotePASS [Alert Status Notify (0 = disable, 1 = enable)]
                           [Ringer Setting Notify (0 = disable, 1 = enable)]
```

Figure 17-6. PASP Demo Configure Remote PASS

4. In this case, both Alert Status Notify and Ringer Setting Notify are enabled by issuing the ConfigureRemotePASS <Alert status notify> <Ringer Setting Notify>

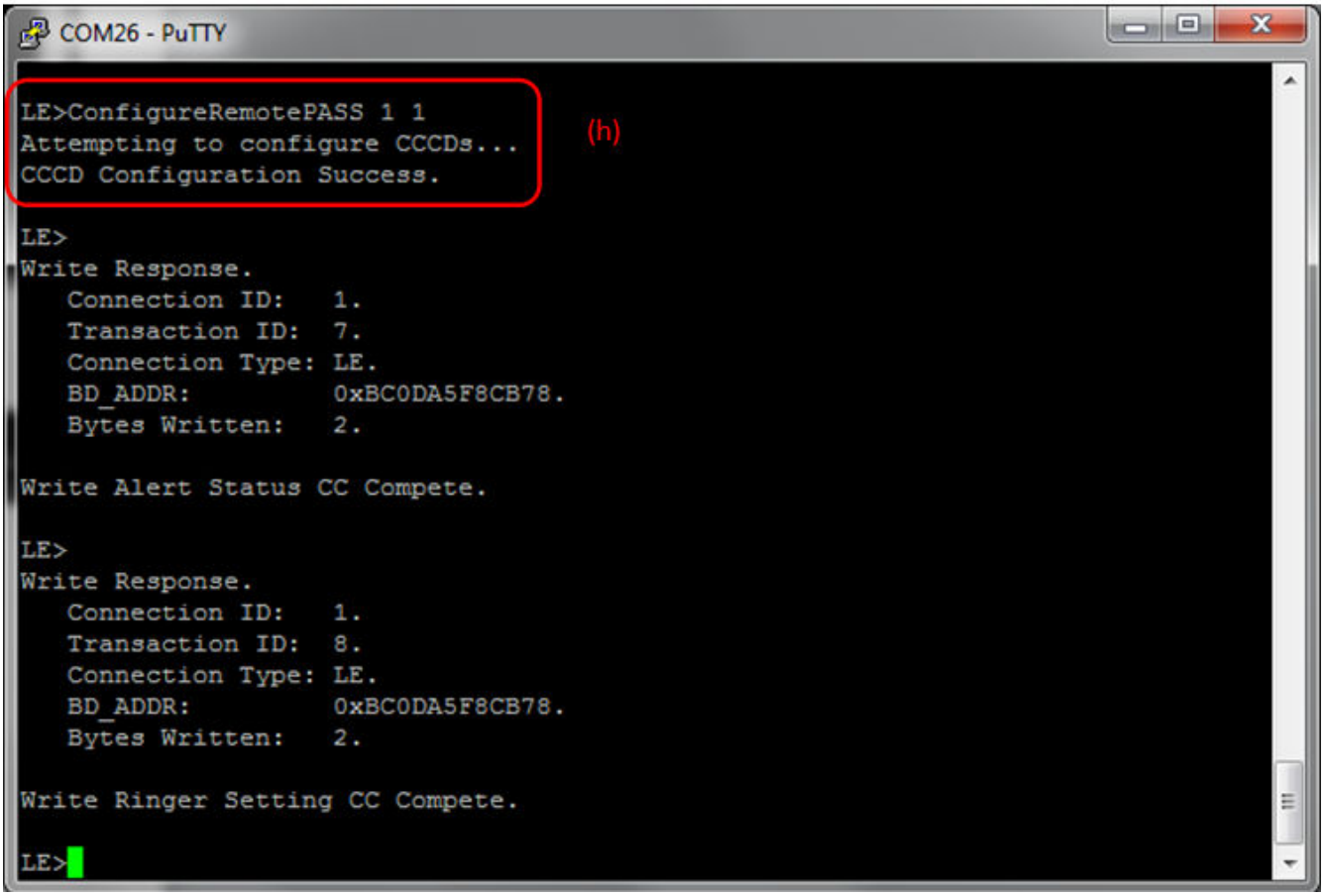

Figure 17-7. PASP Demo Configure Remote PASS Result 2

Alert Notification between Client and Server

1. After configuring, The Phone Alert System is now active. We can get the status of the Alert and Ringer by issuing the NotifyAlertStatus and NotifyRingerSetting commands.

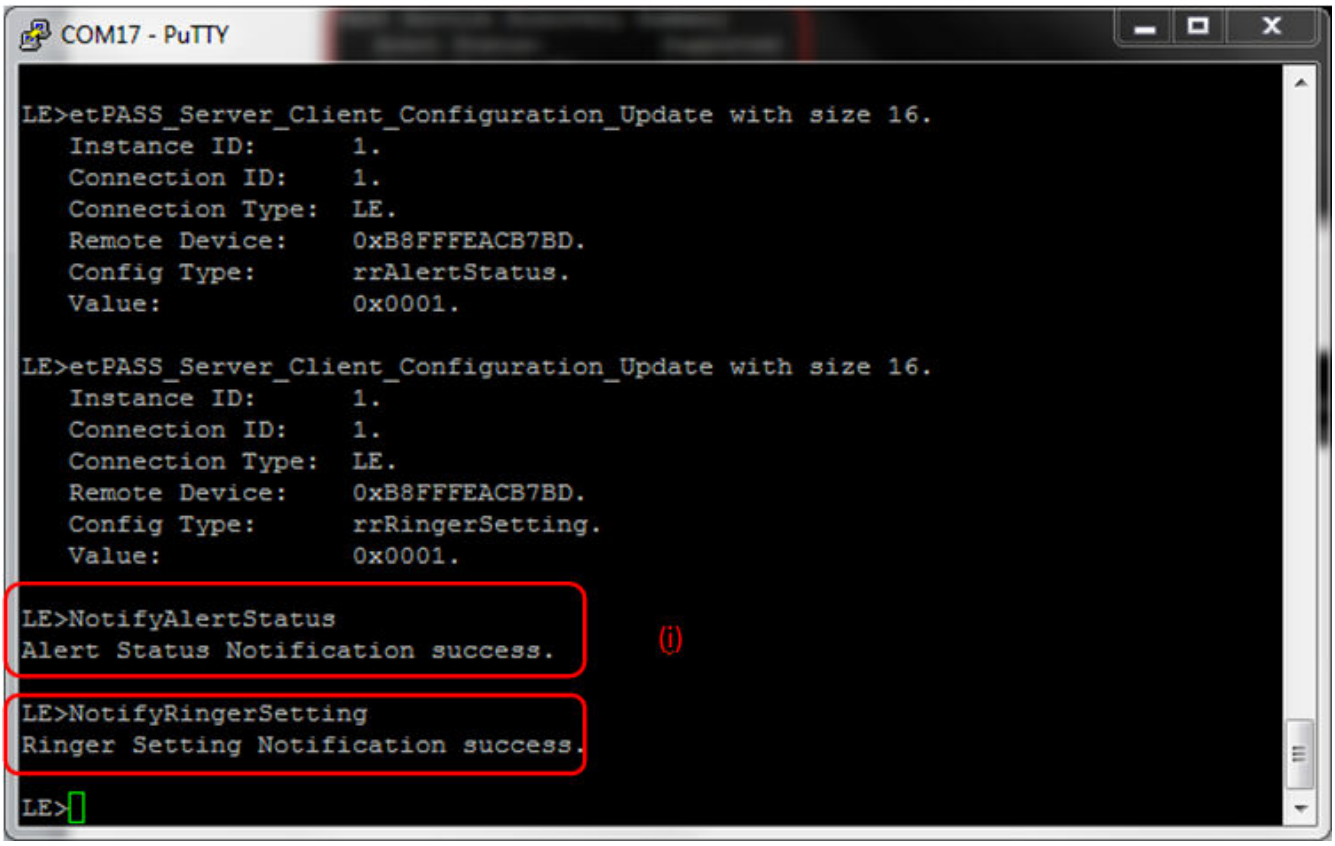

Figure 17-8. PASP Server Run NotifyAlertStatus and NotifyRingerSetting

After running NotifyAlertStatus and NotifyRingerSetting on the Server, we get the Alert Status and Ringer Setting on the Client.

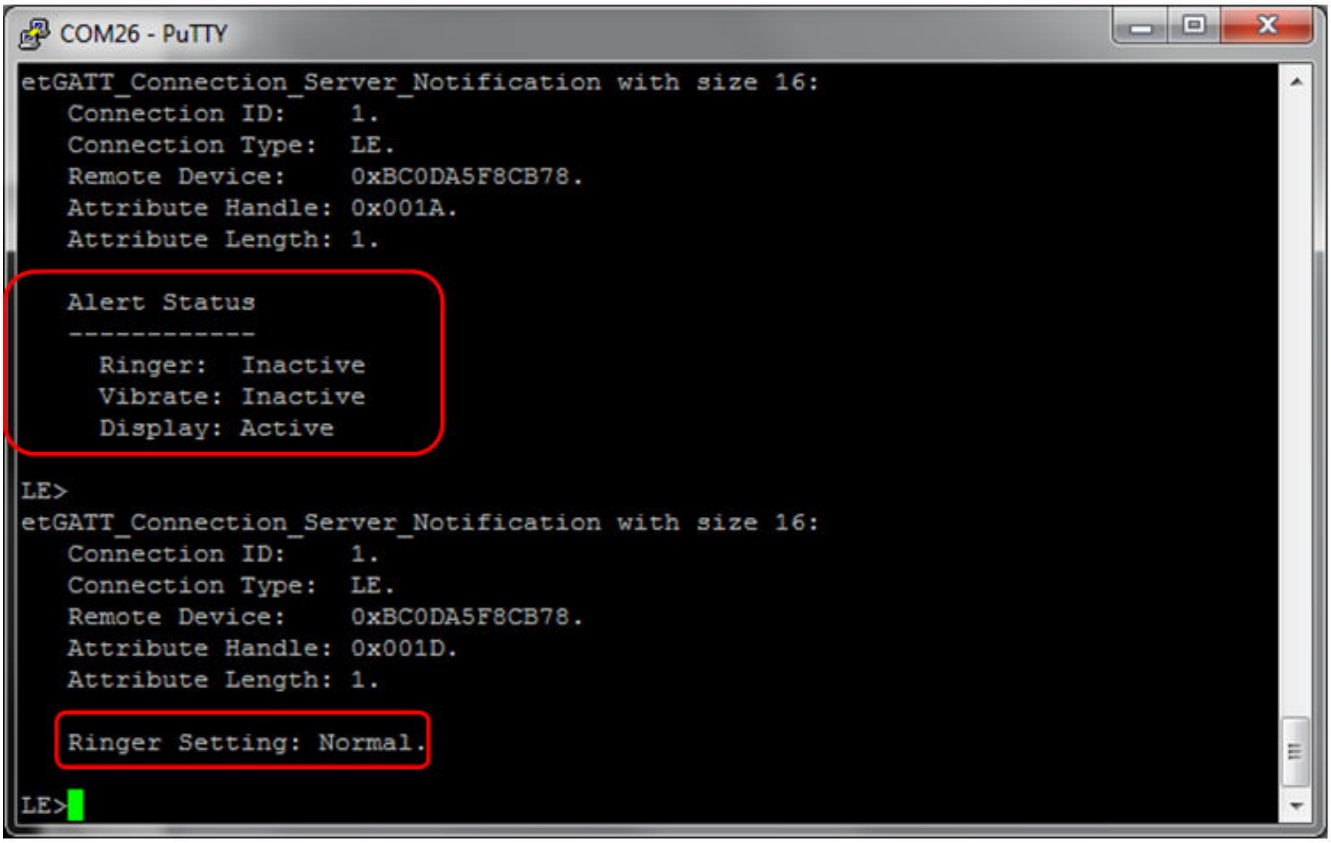

Figure 17-9. PASP Demo Alert Status Notification

2. We can set the Alert Status for each of the three options (Display, Vibrate and Ringer) by issuing the SetAlertStatus <Ringer> <Vibrate> <Display> command and setting each option either active or inactive.

```
LE>SetAlertStatus
Usage: SetAlertStatus [Ringer   (0 = inactive, 1 = active)]
                      [Vibrate  (0 = inactive, 1 = active)]
                      [Display  (0 = inactive, 1 = active)]
```

Figure 17-10. PASP Demo Set Alert Status

In this example, we set all of them to active and then issue the NotifyAlertStatus command.

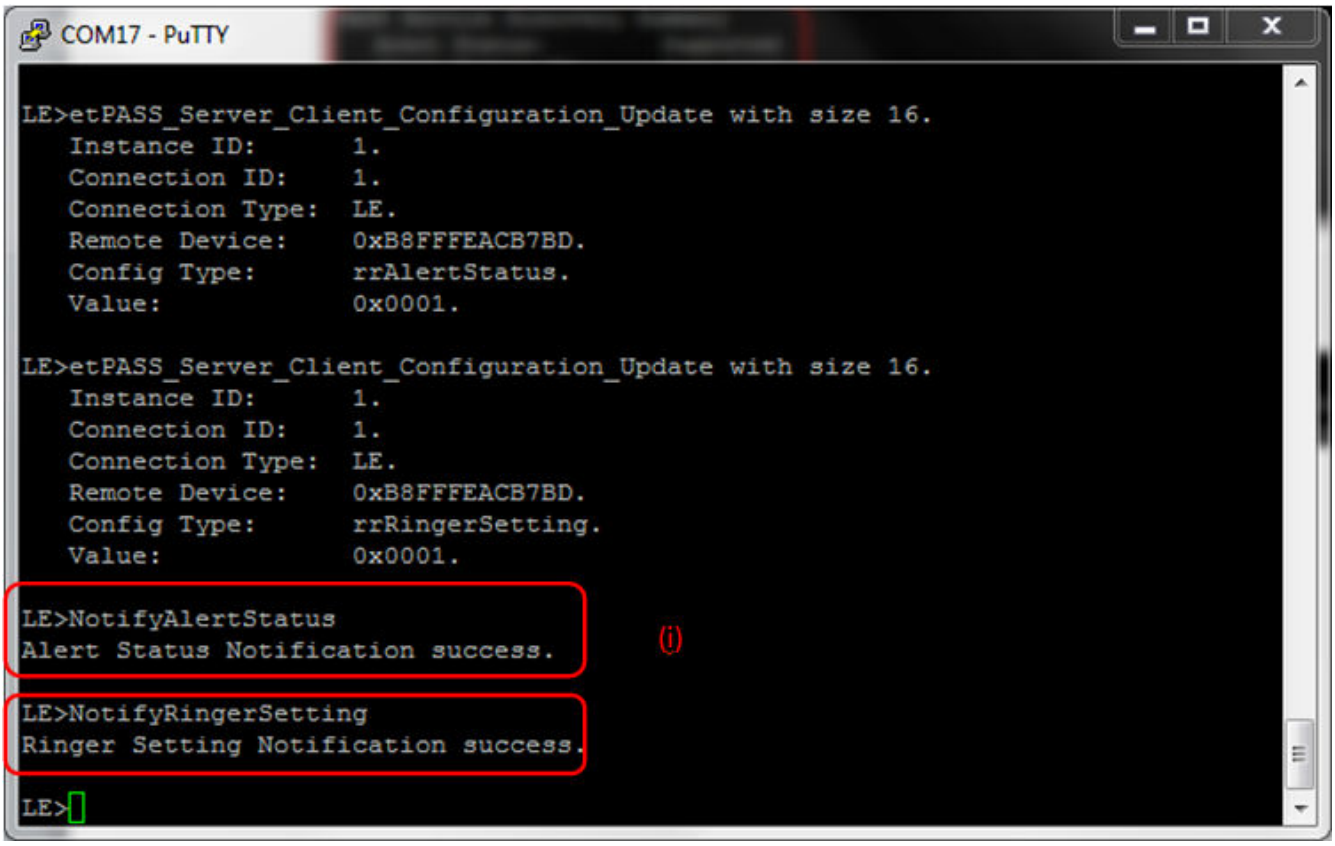

Figure 17-11. PASP Demo Set Alert Status Result

The Client will then receive the updated alert status.

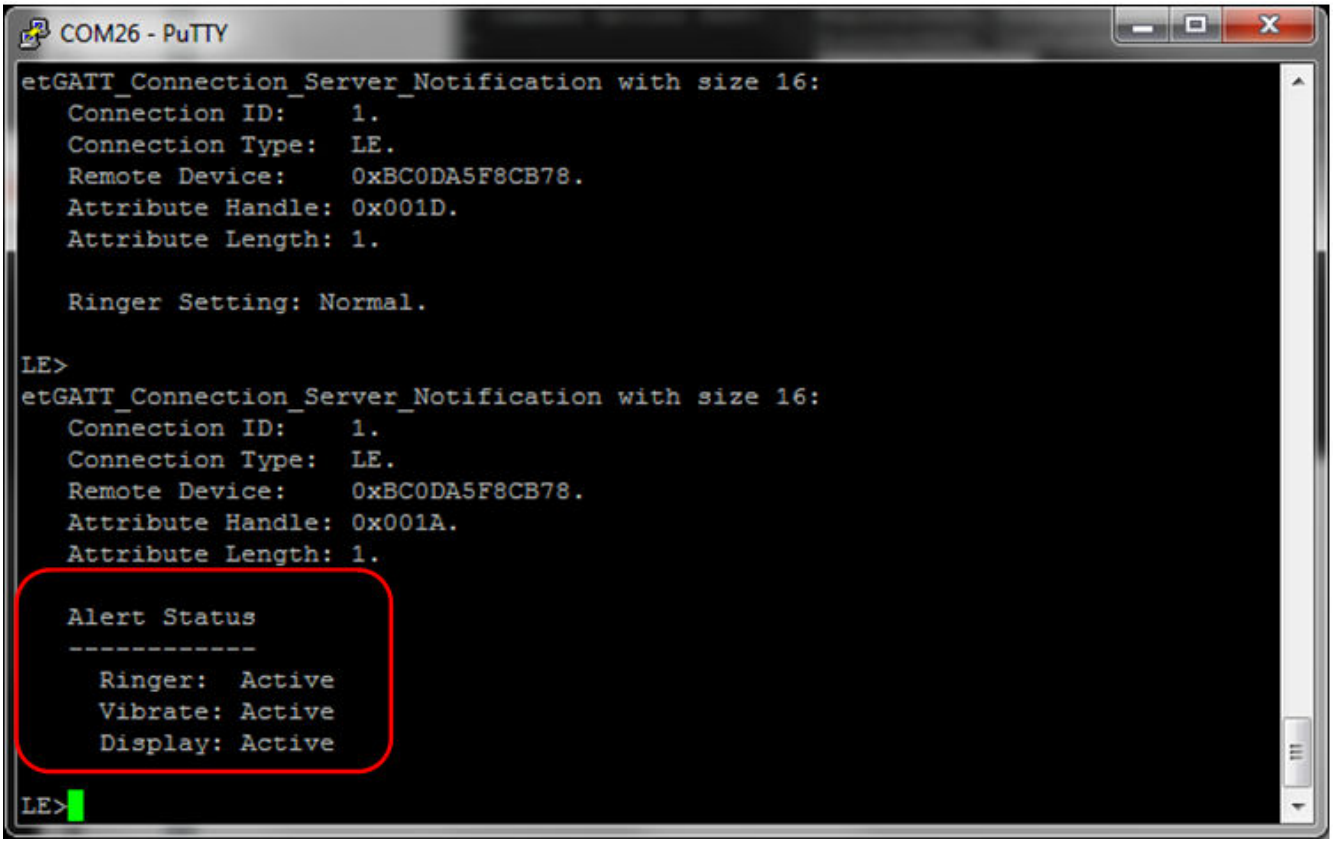

Figure 17-12. PASP Client Receive Updated Alert Status

3. Similarly, we can toggle the ringer setting between Silent and Normal through SetRingerSetting.

```
LE>SetRingerSetting
Usage: SetRingerSetting [(0 = silent, 1 = normal)].
```

Figure 17-13. PASP Demo Set Ringer Setting Usage

In our example, we set toggle between Normal and Silent settings and update the setting by issuing the NotifyRingerSetting command.

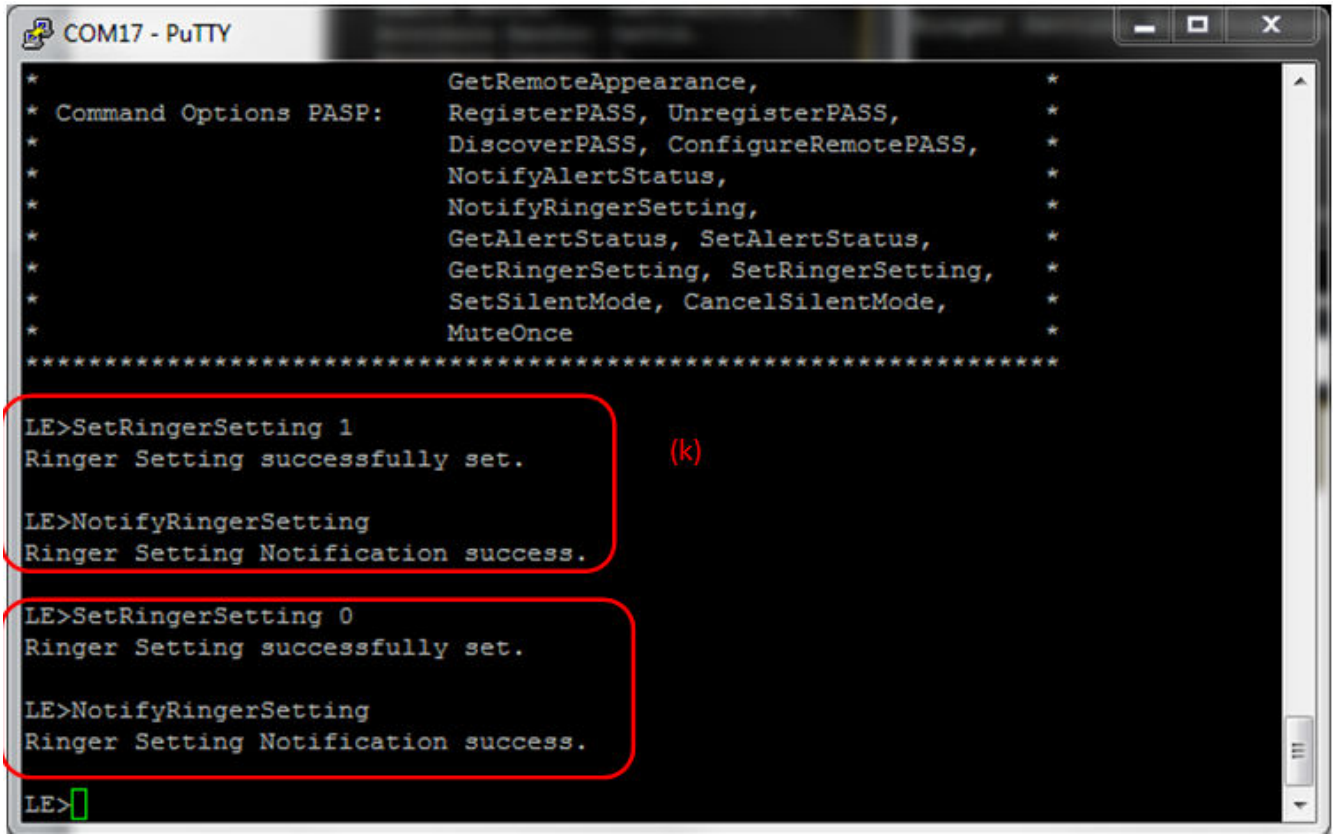

Figure 17-14. PASP Demo Set Ringer Setting

The Client sees the updated Ringer settings.

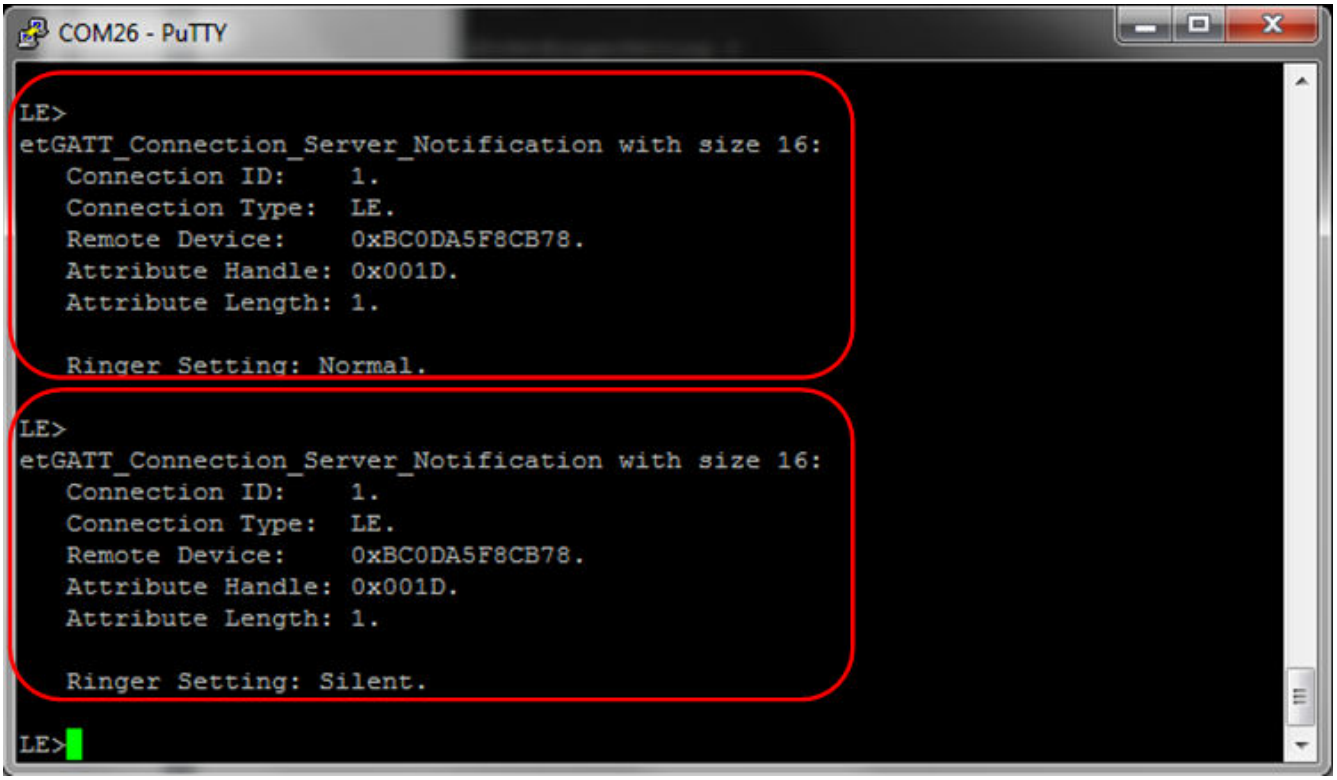

Figure 17-15. PASP Demo Connection Server Notification

We can also move the device into and out of silent mode by issuing the SetSilentMode and CancelSilentMode commands and Mute the device by issuing the MuteOnce command.

17.3 Application Commands

RegisterPASS

Description

This function is responsible for registering a PASP Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully registered PASP Service

(-4) FUNCTION_ERROR

(-1000)PASS_ERROR_INVALID_PARAMETER

(-1002)PASS_ERROR_INSUFFICIENT_RESOURCES

(-1003)PASS_ERROR_SERVICE_ALREADY_REGISTERED

API Call

PASS_Initialize_Service(BluetoothStackID, PASS_EventCallback, NULL, &PASSInstanceID)

API Prototype

int BTPSAPI PASS_Initialize_Service(unsigned int BluetoothStackID, PASS_Event_Callback_t EventCallback, unsigned long CallbackParameter, unsigned int *ServiceID)

Description of API

The following function is responsible for opening a PASS Server. The first parameter is the Bluetooth Stack ID on which to open the Server. The second parameter is the Callback function to call when an event occurs on this Server Port. The third parameter is a user-defined callback parameter that will be passed to the callback function with each event. The final parameter is a pointer to store the GATT Service ID of the registered PASS service. This can be used to include the service registered by this call. This function returns the positive, non-zero, Instance ID or a negative error code.

UnRegisterPASS

Description

This function is responsible for unregistering a PASP Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully unregistered PASP Service

(-4) FUNCTION_ERROR

(-1000)PASS_ERROR_INVALID_PARAMETER

(-1004)PASS_ERROR_INVALID_INSTANCE_ID

API Call

PASS_Cleanup_Service(BluetoothStackID, PASSInstanceID)

API Prototype

int BTPSAPI PASS_Cleanup_Service(unsigned int BluetoothStackID, unsigned int InstanceID)

Description of API

The following function is responsible for closing a previously PASS Server. The first parameter is the Bluetooth Stack ID on which to close the Server. The second parameter is the InstanceID that was returned from a successful call to PASS_Initialize_Service(). This function returns a zero if successful or a negative return error code if an error occurs.

DiscoverPASS

Description

This function is responsible for performing a PASP Service Discovery Operation. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) GDIS_Service_Discovery_Start success

(-4) Function_Error

(-1000)GDIS_ERROR_INVALID_PARAMETER

(-1001)GDIS_ERROR_NOT_INITIALIZED

(-1002)GDIS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003)GDIS_ERROR_INSUFFICIENT_RESOURCES

(-1009)GDIS_ERROR_SERVICE_DISCOVERY_OUTSTANDING

API Call

GDIS_Service_Discovery_Start(BluetoothStackID, ConnectionID, (sizeof(UUID)/sizeof(GATT_UUID_t)), UUID, GDIS_Event_Callback, sdPASS)

API Prototype

int BTPSAPI GDIS_Service_Discovery_Start(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GDIS_Event_Callback_t ServiceDiscoveryCallback, unsigned long ServiceDiscoveryCallbackParameter)

Description of API

The following function is used to initiate the Service Discovery process or queue additional requests. The function takes as its first parameter the BluetoothStackID that is associated with the Bluetooth Device. The second parameter is the connection ID of the remote device that is to be searched. The third and fourth parameters specify an optional list of UUIDs to search for. The final two parameters define the Callback function and parameter to use when the service discovery is complete. The function returns zero on success and a negative return value if there was an error.

ConfigureRemotePASS

Description

This function is responsible for configure a PASP Service on a remote device. This function will return zero on successful execution and a negative value on errors.

Parameters

The ConfigureRemotePASS command requires 2 parameters. The first parameter is Alert Status Notify (0 = disable, 1 = enable) and the second parameter is Ringer Setting Notify (0 = disable, 1 = enable).

Command Call Examples

ConfigureRemotePASS 1 1(both Alert Status Notify and Ringer Setting Notify enabled).

ConfigureRemotePASS 0 0(both Alert Status Notify and Ringer Setting Notify disabled).

ConfigureRemotePASS 0 1(Alert Status Notify disabled and Ringer Setting Notify enabled).

ConfigureRemotePASS 1 0 (Alert Status Notify enabled and Ringer Setting Notify disabled).

Possible Return Values

(0) CCCD Configuration Success

(-1)BTPS_ERROR_INVALID_PARAMETER

(-4) Function_Error

API Call

EnableDisableNotificationsIndications(DeviceInfo->ClientInfo.Alert_Status_Client_Configuration, (TempParam->Params[0].intParam ? GATT_CLIENT_CONFIGURATION_CHARACTERISTIC_NOTIFY_ENABLE : 0), GATT_ClientEventCallback_PASP)

API Prototype

int EnableDisableNotificationsIndications(Word_t ClientConfigurationHandle, Word_t ClientConfigurationValue, GATT_Client_Event_Callback_t ClientEventCallback)

Description of API

The following function function is used to enable/disable notifications on a specified handle. This function returns the positive non-zero Transaction ID of the Write Request or a negative error code.

NotifyAlertStatus

Description

This function is responsible for performing an Alert Status notification to a connected remote device. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0)Alert Status Notification success

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000)PASS_ERROR_INVALID_PARAMETER

(-1004)PASS_ERROR_INVALID_INSTANCE_ID

(-1006)PASS_ERROR_UNKNOWN_ERROR

API Call

PASS_Send_Notification(BluetoothStackID, PASSInstanceID, ConnectionID, rrAlertStatus)

API Prototype

int BTPSAPI PASS_Send_Notification(unsigned int BluetoothStackID, unsigned int InstanceID, unsigned int ConnectionID, PASS_Characteristic_Type_t CharacteristicType)

Description of API

The following function is responsible for sending a notification of a specified characteristic to a specified remote device. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the

InstanceID returned from a successful call to PASS_Initialize_Server(). The third parameter is the ConnectionID of the remote device to send the notification to. The final parameter specifies the characteristic to notify. This function returns a zero if successful or a negative return error code if an error occurs.

NotifyRingerSetting

Description

This function is responsible for performing an Ringer Setting notification to a connected remote device. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Ringer Setting Notification success

(-4) FUNCTION_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000)PASS_ERROR_INVALID_PARAMETER

(-1004)PASS_ERROR_INVALID_INSTANCE_ID

API Call

PASS_Send_Notification(BluetoothStackID, PASSInstanceID, ConnectionID, rrRingerSetting)

API Prototype

int BTPSAPI PASS_Send_Notification(unsigned int BluetoothStackID, unsigned int InstanceID, unsigned int ConnectionID, PASS_Characteristic_Type_t CharacteristicType)

Description of API

The following function is responsible for sending a notification of a specified characteristic to a specified remote device. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to PASS_Initialize_Server(). The third parameter is the ConnectionID of the remote device to send the notification to. The final parameter specifies the characteristic to notify. This function returns a zero if successful or a negative return error code if an error occurs.

GetAlertStatus

Description

This function is responsible for reading the Alert Status. It can be executed by a Server or a Client with an open connection to a remote Server. If executed as a Client, a GATT read request will be generated, and the results will be returned as a response in the GATT Client event callback. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Get Alert Status Request sent

(-4) FUNCTION_ERROR

API Call

GATT_Read_Value_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.Alert_Status, GATT_ClientEventCallback_PASP, DeviceInfo->ClientInfo.Alert_Status)

API Prototype

int BTPSAPI GATT_Read_Value_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, GATT_Client_Event_Callback_t ClientEventCallback, unsigned long CallbackParameter)

Description of API

The following function is provided to allow a means of performing a read request on a remote device for a specific attribute value. The first parameter is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the attribute handle to read the value from. The final two parameters specify the GATT Client event callback function and callback parameter (respectively) that will be called when a response is received from the remote device. This function will return the positive, non-zero, Transaction ID of the request or a negative error code.

SetAlertStatus

Description

This function is responsible for writing the Alert Status. It can be executed only by a Server. This function will return zero on successful execution and a negative value on errors.

Parameters

Three parameters are required for this function. The first one is Ringer (0 = inactive, 1 = active). The second one is Vibrate (0 = inactive, 1 = active) and the third one is Display (0 = inactive, 1 = active)

Command Call Examples

SetAlertStatus 1 1 1 (Sets Ringer, Vibrate and Display to Active)

SetAlertStatus 0 1 1 (Sets Ringer to Inactive, Vibrate and Display to Active)

SetAlertStatus 1 1 0 (Sets Ringer, Vibrate to Active and Display to Inactive)

Possible Return Values

(0) Alert Status successfully set

(-4) FUNCTION_ERROR

(-1000)PASS_ERROR_INVALID_PARAMETER

(-1004)PASS_ERROR_INVALID_INSTANCE_ID

API Call

PASS_Set_Alert_Status(BluetoothStackID, PASSInstanceID, AlertStatus)

API Prototype

int BTPSAPI PASS_Set_Alert_Status(unsigned int BluetoothStackID, unsigned int InstanceID, PASS_Alert_Status_t AlertStatus)

Description of API

The following function is responsible for setting the Alert Status characteristic on the specified PASS instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to PASS_Initialize_Server(). The final parameter is the Alert Status to set as the current Alert Status for the specified PASS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

GetRingerSetting

Description

This function is responsible for reading the Ringer Setting. It can be executed by a Server or a Client with an open connection to a remote Server. If executed as a Client, a GATT read request will be generated, and the results will be returned as a response in the GATT Client event callback. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Ringer Setting

(-4) FUNCTION_ERROR

(-1000)PASS_ERROR_INVALID_PARAMETER

(-1004)PASS_ERROR_INVALID_INSTANCE_ID

API CallPASS_Query_Ringer_Setting(BluetoothStackID, PASSInstanceID, &RingerSetting)

API Prototype

int BTPSAPI PASS_Query_Ringer_Setting(unsigned int BluetoothStackID, unsigned int InstanceID, PASS_Ringer_Setting_t *RingerSetting)

Description of API

The following function is responsible for querying the current Ringer Setting characteristic value on the specified PASS instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to PASS_Initialize_Server(). The final parameter is a pointer to store the current Ringer Setting for the specified PASS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

SetRingerSetting

Description

This function is responsible for writing the Ringer Setting. It can be executed only by a Server. This function will return zero on successful execution and a negative value on errors.

Parameters

SetRingerSetting has only one parameter which is an integer value that represents a Ringer Setting. This value is 0 = silent, 1 = normal.

Possible Return Values

(0) Ringer Setting successfully set

(-4) FUNCTION_ERROR

(-1000)PASS_ERROR_INVALID_PARAMETER

(-1004)PASS_ERROR_INVALID_INSTANCE_ID

API Call

PASS_Set_Ringer_Setting(BluetoothStackID, PASSInstanceID, RingerSetting)

API Prototype

int BTPSAPI PASS_Set_Ringer_Setting(unsigned int BluetoothStackID, unsigned int InstanceID, PASS_Ringer_Setting_t RingerSetting)

Description of API

The following function is responsible for setting the Ringer Setting characteristic on the specified PASS instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to PASS_Initialize_Server(). The final parameter is the Ringer Setting to set as the current Ringer Setting for the specified PASS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

SetSilentMode

Description

This function is responsible for writing the Set Silent Mode command to a remote Server Control Point. It can be executed only by a Client. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0)Set Silent Mode command request sent

(-4) FUNCTION_ERROR

API Call

GATT_Write_Without_Response_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.Ringer_Control_Point, PASS_RINGER_CONTROL_POINT_VALUE_LENGTH, ((void *)CommandBuffer)

API Prototype

int BTPSAPI GATT_Write_Without_Response_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue)

Description of API

The following function is provided to allow a means of performing a write without response request to remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write, followed by the length of the value data to write (in bytes), followed by the actual value to write. This function will return the number of bytes written on success or a negative error code.

CancelSilentMode

Description

This function is responsible for writing the Cancel Silent Mode command to a remote Server Control Point. It can be executed only by a Client. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0)Cancel Silent Mode command request sent

(-4) FUNCTION_ERROR

API Call

GATT_Write_Without_Response_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.Ringer_Control_Point, PASS_RINGER_CONTROL_POINT_VALUE_LENGTH, ((void *)CommandBuffer)

API Prototype

int BTPSAPI GATT_Write_Without_Response_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue)

Description of API

The following function is provided to allow a means of performing a write without response request to remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute

to write, followed by the length of the value data to write (in bytes), followed by the actual value to write. This function will return the number of bytes written on success or a negative error code.

MuteOnce

Description

This function is responsible for writing the Mute Once command to a remote Server Control Point. It can be executed only by a Client. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0)Mute Once command request sent

(-4) FUNCTION_ERROR

API Call

GATT_Write_Without_Response_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.Ringer_Control_Point, PASS_RINGER_CONTROL_POINT_VALUE_LENGTH, ((void *)CommandBuffer)))

API Prototype

int BTPSAPI GATT_Write_Without_Response_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue)

Description of API

The following function is provided to allow a means of performing a write without response request to remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write, followed by the length of the value data to write (in bytes), followed by the actual value to write. This function will return the number of bytes written on success or a negative error code.

18 HOGP Demo Guide

18.1 Demo Overview

Note

The same instructions can be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms.

The HID Over GATT Profile (HOGP) allows a user to use HID services using Bluetooth Low Energy.

It is recommended that the user visits the kit setup Getting Started Guide page before trying the application described on this page.

Note

For the MSP430 version of this demo, please go to here

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as Tiva Virtual COM Port (COM x) or XDS110 Class Application/User UART (COM x) for MSP432 under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud (9600 for MSP430), 8, n, 1. Once connected, reset the device using Reset S3 button and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

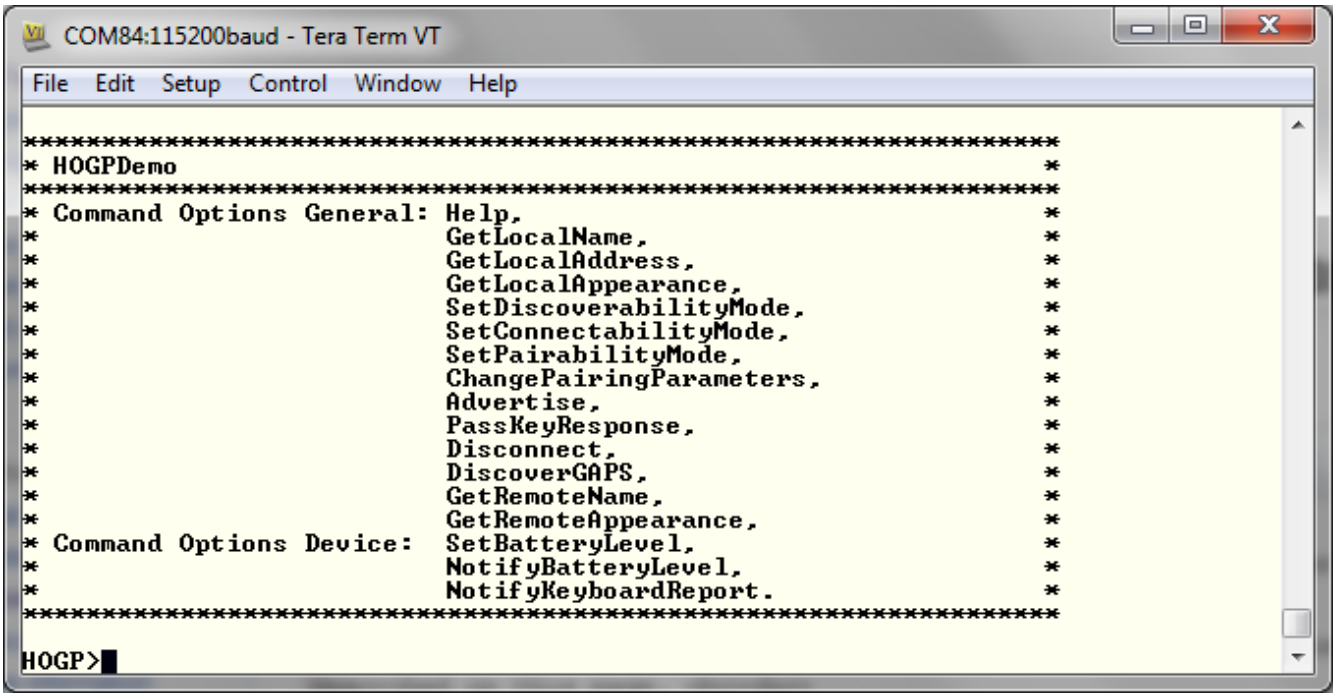

Figure 18-1. HOGP Demo Start

18.2 Demo Application

This section provides a description of how to use the demo application to connect an iPhone with the HOGP application and communicate over Bluetooth.

Demo application Setup

1. One command is needed to setup the server. Begin advertising by typing Advertise 1.

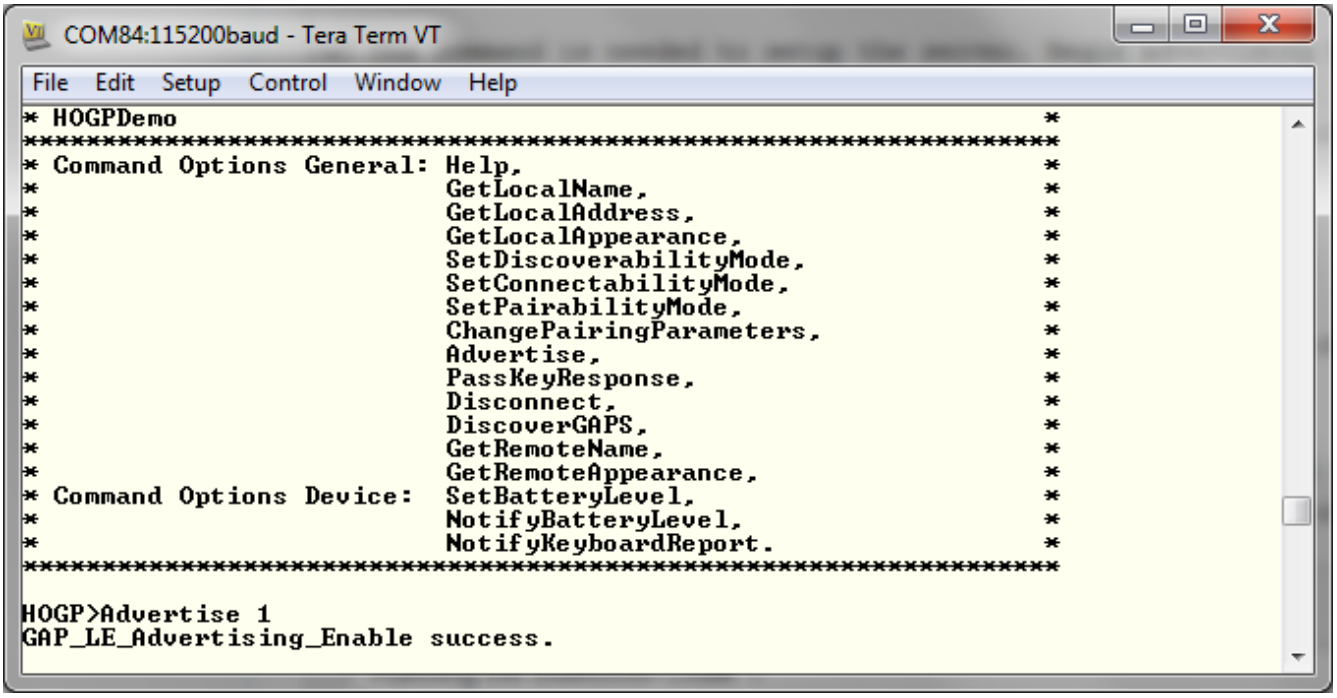

Figure 18-2. HOGP Demo Advertise Start

iPhone Setup

1. Turn on Bluetooth and begin searching for devices.

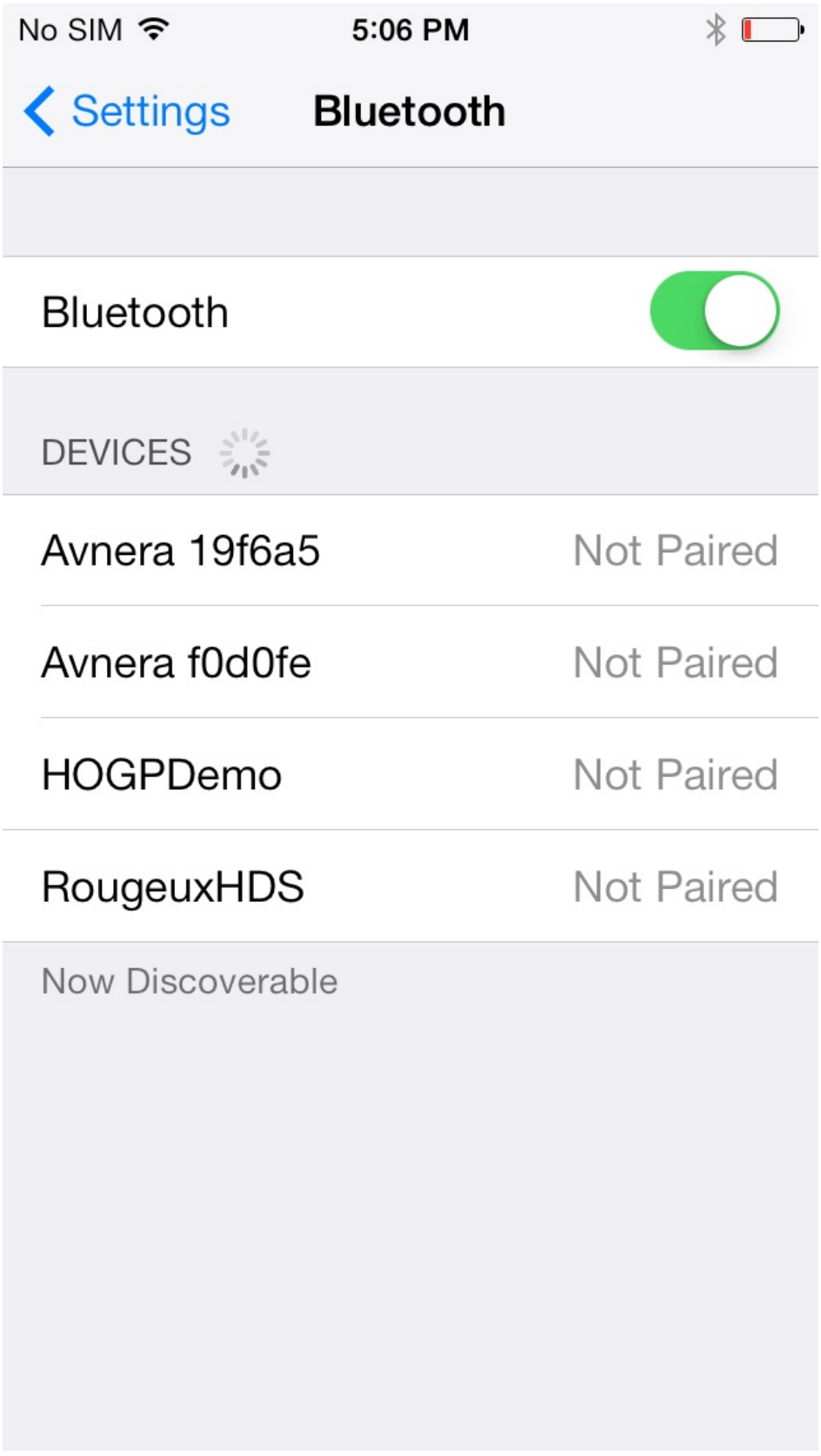

Figure 18-3. HOGP Demo Device Discovery

Initiating connection from the iPhone

1. Look for a device named HOGPdemo on the iPhone and pair with it. You should see pairing indications on both sides.

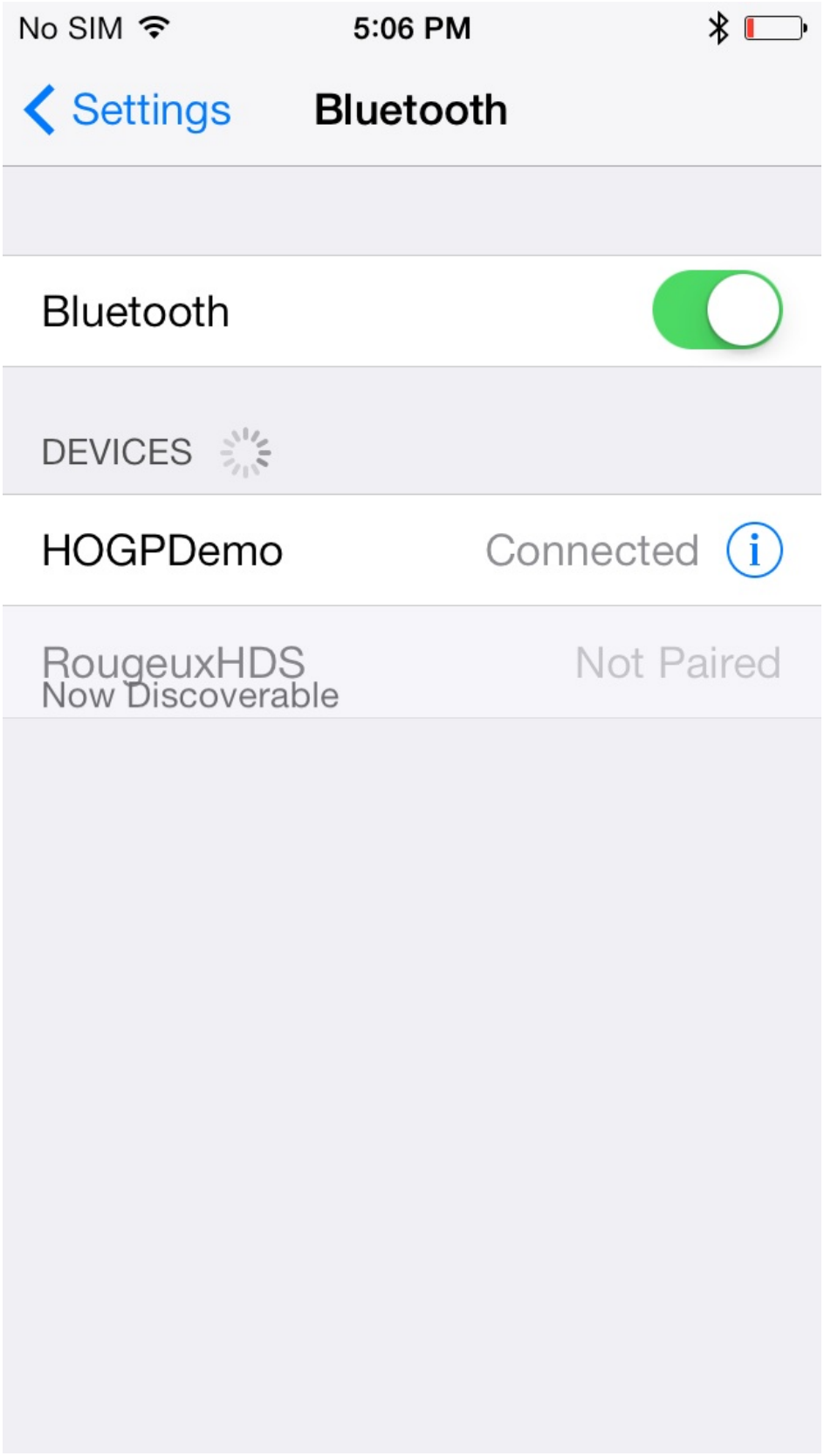

Figure 18-4. HOGP Demo Device Connected

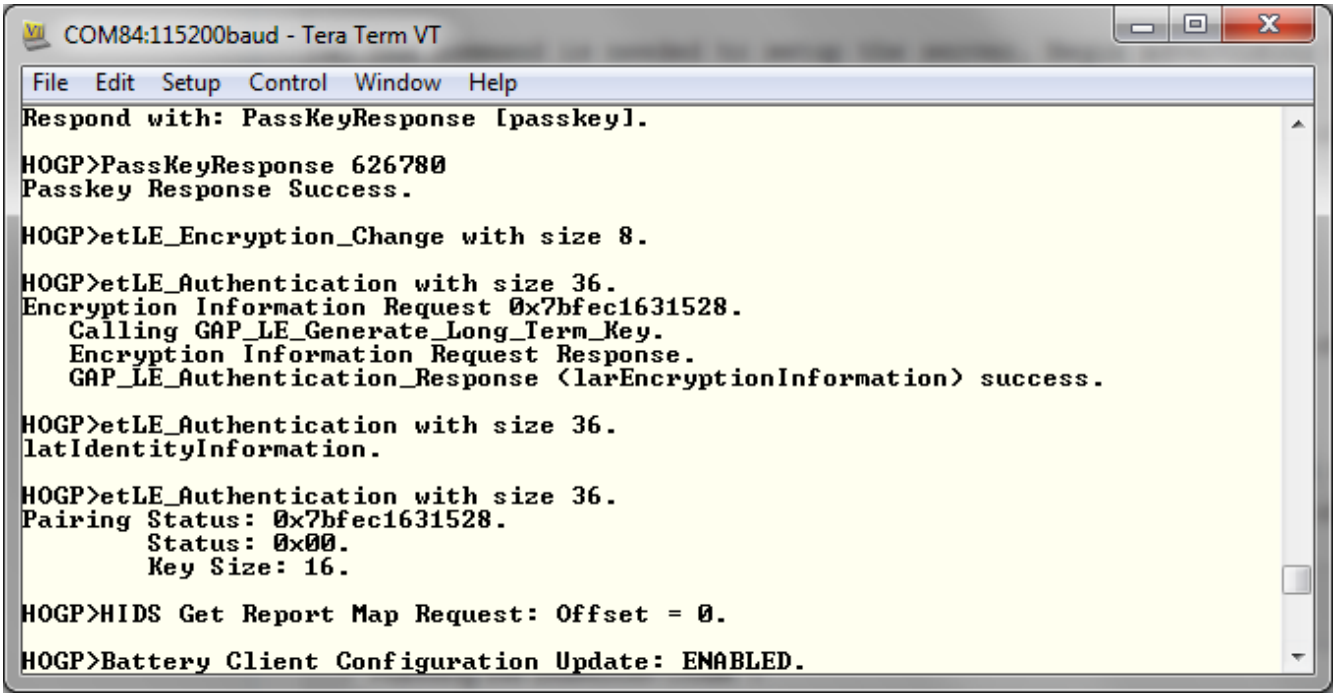

Figure 18-5. HOGP Demo Device Connection Complete Remote Side

Data Transfer between Iphone and Device

1. Open any text editor on the iphone and from the device enter the NotifyKeyboardReport keystroke command. The iPhone should receive it.

```
HOGP>
NotifyKeyboardReport 4

HOGP>NotifyKeyboardReport 1

HOGP>NotifyKeyboardReport 1
```

Figure 18-6. HOGP NotifyKeyboardReport

Figure 18-7. HOGP Demo Data Receive

18.3 Application Commands

SetBatteryLevel

Description

The following function is responsible for setting the Battery Level of the LE device. This function takes an unsigned integer as a parameter for the desired battery level and returns zero for success or a negative number indicating an error.

Parameters

The only parameter required is battery level.

Command Call Examples

SetBatteryLevel 100 attempts to set the battery level to 100.

Possible Return Values

(0) MAP_Close_Server() Success

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

NotifyBatteryLevel

Description

The following function is responsible for notifying the battery level to the connected host device (if the device has registered for notifications). This function takes no parameters and returns zero on success or a negative number indicating an error.

Parameters

The only parameter required is battery level.

Command Call Examples

NotifyBatteryLevel 100 attempts to notify the battery level to 100.

Possible Return Values

(0) MAP_Close_Server() Success

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) BAS_ERROR_INVALID_PARAMETER

(-1001) BAS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1002) BAS_ERROR_INSUFFICIENT_RESOURCES

(-1003) BAS_ERROR_SERVICE_ALREADY_REGISTERED

(-1004) BAS_ERROR_INVALID_INSTANCE_ID

(-1005) BAS_ERROR_MALFORMATTED_DATA

(-1006) BAS_ERROR_MAXIMUM_NUMBER_OF_INSTANCES_REACHED

(-1007) BAS_ERROR_UNKNOWN_ERROR

API Call

BAS_Notify_Battery_Level(ApplicationStateInfo.BluetoothStackID, ApplicationStateInfo.BASInstanceID, ApplicationStateInfo.LEConnectionInfo.ConnectionID, (Byte_t)ApplicationStateInfo.BatteryLevel);

API Prototype

int BTPSAPI BAS_Notify_Battery_Level(unsigned int BluetoothStackID, unsigned int InstanceID, unsigned int ConnectionID, Byte_t BatteryLevel);

Description of API

The following function is responsible for sending a Battery Level Status notification to a specified remote device. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to BAS_Initialize_Server(). The third parameter is the ConnectionID of the remote device to send the notification to. The final parameter contains the Battery Level to send to the remote device. This function returns a zero if successful or a negative return error code if an error occurs.

NotifyKeyboardReport

Description

The following function is responsible for notifying a report to the connected host device (if the device has registered for notifications). The function takes a string parameter that contains an alpha-numeric character that should be included in the report to be notified. This function returns zero on success or a negative number indicating an error.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the server closing.

Possible Return Values

(0) MAP_Close_Server() Success

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000)HIDS_ERROR_INVALID_PARAMETER (-1001)HIDS_ERROR_INVALID_BLUETOOTH_STACK_ID (-1002)HIDS_ERROR_INSUFFICIENT_RESOURCES (-1003)HIDS_ERROR_SERVICE_ALREADY_REGISTERED (-1004)HIDS_ERROR_INVALID_INSTANCE_ID (-1005)HIDS_ERROR_MALFORMATTED_DATA (-1006)HIDS_ERROR_INDICATION_OUTSTANDING (-1007)HIDS_ERROR_UNKNOWN_ERROR

API Call

HIDS_Notify_Input_Report(ApplicationStateInfo.BluetoothStackID, ApplicationStateInfo.HIDSInstanceID, ApplicationStateInfo.LEConnectionInfo.ConnectionID, rtBootKeyboardInputReport, NULL, HID_KEYBOARD_INPUT_REPORT_SIZE, ApplicationStateInfo.CurrentInputReport);

API Prototype

int BTPSAPI HIDS_Notify_Input_Report(unsigned int BluetoothStackID, unsigned int InstanceID, unsigned int ConnectionID, HIDS_Report_Type_t ReportType, HIDS_Report_Reference_Data_t *ReportReferenceData, Word_t InputReportLength, Byte_t *InputReportData)

Description of API

The following function is responsible for sending an Input Report notification to a specified remote device. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to HIDS_Initialize_Server(). The third parameter is the ConnectionID of the remote device to send the notification to. The fourth parameter specifies the ReportType of the report that is to be notified. The fifth parameter is a pointer to a Report Reference structure that is only used (and must be specified only if) the ReportType is rtInputReport. The final parameters contain the length of the Input Report and a pointer to the Input Report that is to be notified. This function returns the number of bytes that were successfully notified on success or a negative return error code if an error occurs.

Note

The only valid values that ReportType may be set to are as follows:
rtReport (Input Report Only) rtBootKeyboardInputReport rtBootMouseInputReport

Note

If the ReportType is rtReport, then the ReportReferenceData must be valid and point to a Report Reference structure of the Input Report to be notified. Otherwise the ReportReferenceData parameter can be NULL.

19 PXP Demo Guide

19.1 Demo Overview

Note

The same instructions can be used to run this demo on the Tiva, MSP432 or STM32F4 Platforms.

The Proximity profile is used to notify the Client about Link Loss and Power Level of the Server device. There are two roles defined in this profile, Server and the Client. Any changes made to the Power Level or Link Loss is also notified by the Server to the Client.

This application allows the user to use a console to use Bluetooth Low Energy (BLE) to establish connection between two BLE devices, Notify Link Loss, Set Alert Levels and Get Transmit Power Levels.

It is recommended that the user visits the kit setup Getting Started Guide for MSP430, Getting Started Guide for TIVA, Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as MSP-EXP430F5438 USB - Serial Port (COM x), Tiva Virtual COM Port (COM x), XDS110 Class Application/User UART (COM x) for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud, 8, n, 1. Once connected, reset the device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

```
Bluetooth Stack ID: 1.
Device Chipset: 4.0.
BD_ADDR: 0xbc0da5f8d90e

*****************************************************************
* Command Options General: Help, GetLocalAddress               *
*                          SetDiscoverabilityMode,             *
*                          SetConnectabilityMode,              *
*                          SetPairabilityMode,                 *
*                          ChangePairingParameters,            *
*                          SetRandomAddress,                   *
*                          ResolveRemoteAddress,               *
*                          AdvertiseLE, StartScanning,         *
*                          StopScanning, ConnectLE,            *
*                          DisconnectLE, PairLE,               *
*                          LEPasskeyResponse,                  *
*                          QueryEncryptionMode, SetPasskey,    *
*                          DiscoverGAPS, GetLocalName,         *
*                          SetLocalName, GetRemoteName,        *
*                          SetLocalAppearance,                 *
*                          GetLocalAppearance,                 *
*                          GetRemoteAppearance,                *
*                          GetMTU, SetMTU,                     *
* Command Options LLS:     RegisterLLS, UnRegisterLLS,         *
*                          DiscoverLLS,                        *
*                          GetAlertLevel,                      *
*                          SetAlertLevel,                      *
* Command Options IAS:     RegisterIAS, UnRegisterIAS,         *
*                          DiscoverIAS,                        *
*                          ConfigureAlertLevel,                *
* Command Options TPS:     RegisterTPS, UnRegisterTPS,         *
*                          DiscoverTPS,                        *
*                          SetTxPowerLevel,                    *
*                          GetTxPowerLevel,                    *
*****************************************************************

LE>
```

Figure 19-1. PXP Demo Start

19.2 Demo Application

This demo application page provides a description of how to connect two configured boards and communicate over Bluetooth Low Energy (BLE). The included application registers a custom service on a board when the stack is initialized.

Device 1 (Server) setup on the demo application

1. To start, one of the devices has to have the Link Loss Service (LLS), Immediate Alert Szervice (IAS) and Transmit Power Service (TPS) running on it. It can be started by issuing the RegisterLLS, RegisterIAS and RegisterTPS commands.

```
*************************************************************************
* Command Options General: Help, GetLocalAddress                       *
*                          SetDiscoverabilityMode,                     *
*                          SetConnectabilityMode,                      *
*                          SetPairabilityMode,                         *
*                          ChangePairingParameters,                    *
*                          SetRandomAddress,                           *
*                          ResolveRemoteAddress,                       *
*                          AdvertiseLE, StartScanning,                 *
*                          StopScanning, ConnectLE,                    *
*                          DisconnectLE, PairLE,                       *
*                          LEPasskeyResponse,                          *
*                          QueryEncryptionMode, SetPasskey,            *
*                          DiscoverGAPS, GetLocalName,                 *
*                          SetLocalName, GetRemoteName,                *
*                          SetLocalAppearance,                         *
*                          GetLocalAppearance,                         *
*                          GetRemoteAppearance,                        *
*                          GetMTU, SetMTU,                             *
* Command Options LLS:     RegisterLLS, UnRegisterLLS,                 *
*                          DiscoverLLS,                                *
*                          GetAlertLevel,                              *
*                          SetAlertLevel,                              *
* Command Options IAS:     RegisterIAS, UnRegisterIAS,                 *
*                          DiscoverIAS,                                *
*                          ConfigureAlertLevel,                        *
* Command Options TPS:     RegisterTPS, UnRegisterTPS,                 *
*                          DiscoverTPS,                                *
*                          SetTxPowerLevel,                            *
*                          GetTxPowerLevel,                            *
*************************************************************************

LE>registerlls
Successfully registered LLS Service.

LE>registerias
Successfully registered IAS Service.

LE>registertps
Successfully registered TPS Service.
```

Figure 19-2. PXP Demo Register Services

2. Next, the device acting as a Server needs to advertise to other devices. This can be done by issuing the AdvertiseLE <Enable/Disable> <Address Type> command.

```
LE>RegisterLLS
Successfully registered LLS Service.
LE>RegisterIAS
Successfully registered IAS Service.
LE>RegisterTPS
Successfully registered TPS Service.
LE>AdvertiseLE 1 1
   GAP_LE_Advertising_Enable success.
LE>_
```

Figure 19-3. PXP Demo Register More Services

Device 2 (Client) setup on the demo application

Note

Steps c and d are optional if you already know the Bluetooth address of the device that you want to connect to.

1. The Client LE device can try to find which LE devices are in the vicinity issuing the StartScanning command.
2. Once you have found the device, you can stop scanning by issuing the StopScanning.

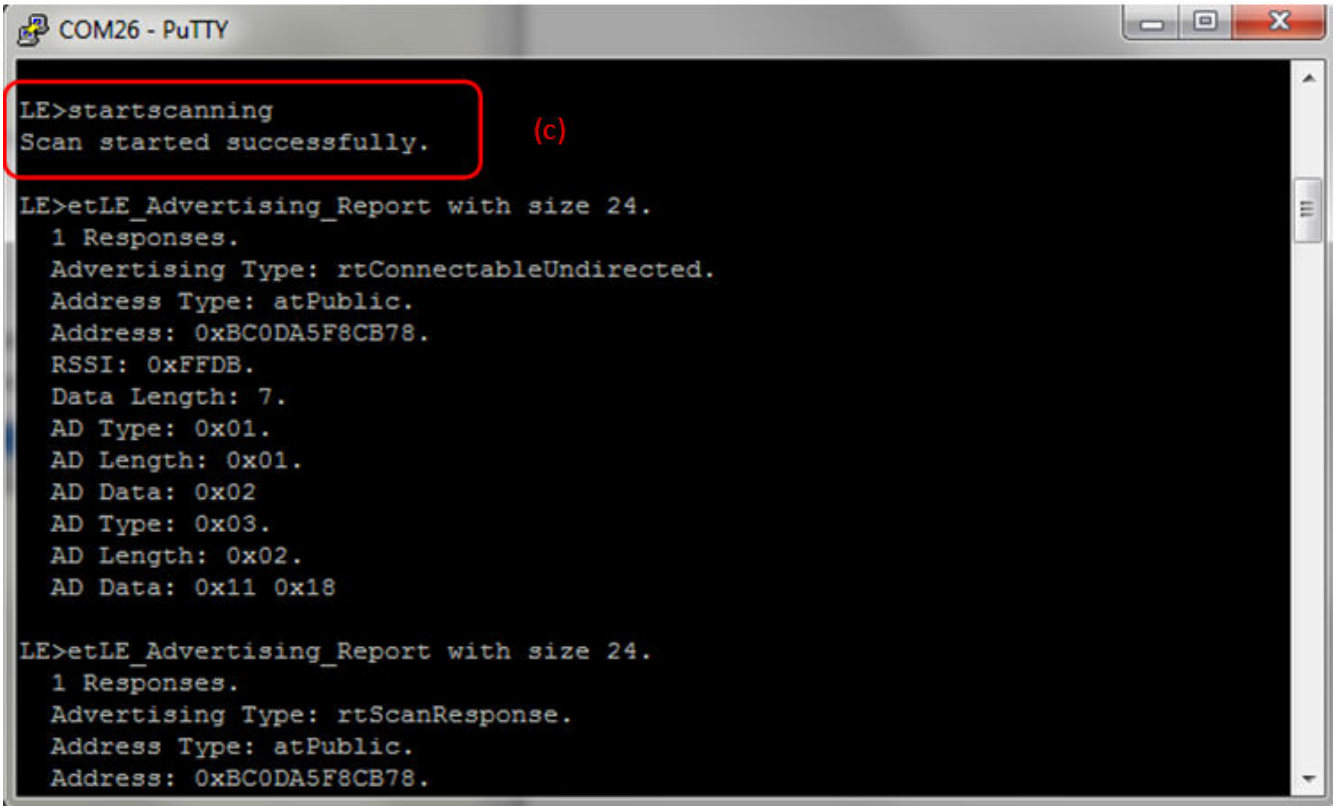

Figure 19-4. PXP Demo Start Scanning

Initiating connection from device 2

1. Once the application on the Client side knows the Bluetooth address of the device that is advertising, it can connect to that device by issuing the ConnectLE <Bluetooth Address of Remote Device> <Remote Address Type> <Own Address Type>

Figure 19-5. PXP Demo Connect BLE

Identify supported services

1. After initialization, the Client needs to find out whether LLS, IAS and TPS services are supported. For this the DiscoverLLS, DiscoverIAS, DiscoverTPS commands are issued on the Client. After the service discovery operations are complete, the LLS, IAS and TPS Service Discovery Summary and list of supported features is shown.

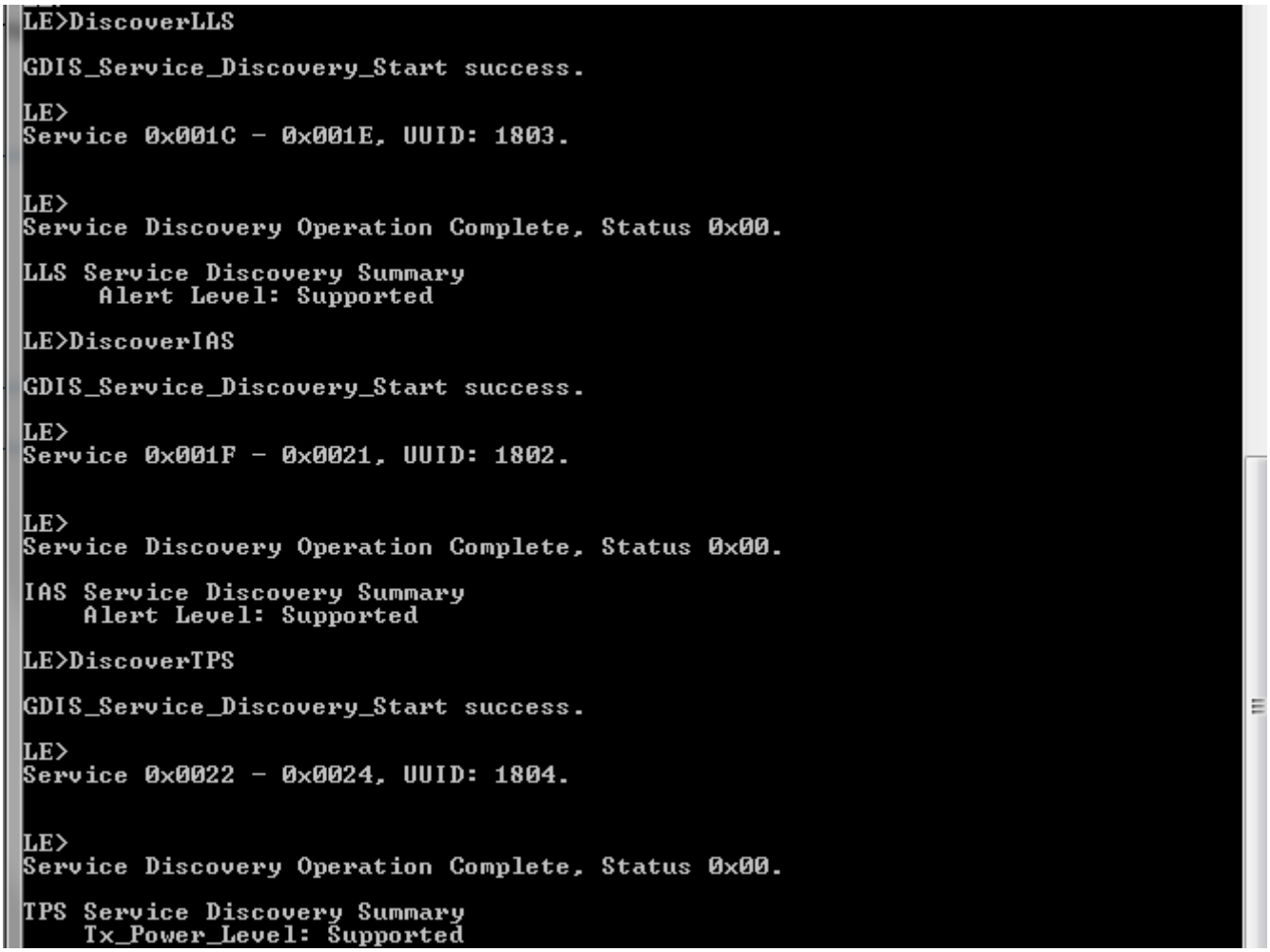

Figure 19-6. PXP Demo Discover Services

Proximity Notification between Client and Server

1. Change the Alert Level from the Client. To change the Alert Level from Client, the commands SetAlertLevel and GetAlertLevel are issued to change and view the updated alert level.

```
LE>SetAlertLevel 2

Set Alert Level Request sent, Transaction ID = 20
LE>
Write Response.
   Connection ID:   3.
   Transaction ID:  20.
   Connection Type: LE.
   BD_ADDR:         0x000272D6A6E2.
   Bytes Written:   1.

LE>GetAlertLevel

Get Alert Level request sent, Transaction ID = 21
LE>
Read Response.
   Connection ID:   3.
   Transaction ID:  21.
   Connection Type: LE.
   BD_ADDR:         0x000272D6A6E2.
   Data Length:     1.

   LLS Alert Level: 2.
```

Figure 19-7. PXP Demo Set Alert Level

2. Change Transmtheit power level (TxPower) from the Client. To change the TxPower Level, the SetTxPowerLevel command is issued from the Server while to obtain the TxPower Level the GetTxPowerLevel command is issued from the Client.

```
LE>SetTxPowerLevel 2

Tx Power Level successfully set.

LE>
```

Figure 19-8. PXP Demo Power Level

```
LE>GetTxPowerLevel

Get Tx Power Level request sent, Transaction ID = 22
LE>
Read Response.
   Connection ID:   3.
   Transaction ID:  22.
   Connection Type: LE.
   BD_ADDR:         0x000272D6A6E2.
   Data Length:     1.

   Tx Power Level: 2.
```

Figure 19-9. PXP Demo Get Power Level

19.3 Applications Commands

RegisterLLS

Description

This function is responsible for registering a LLS Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully registered PASP Service

(-4) FUNCTION_ERROR

(-1000)LLS_ERROR_INVALID_PARAMETER

(-1002)LLS_ERROR_INSUFFICIENT_RESOURCES

(-1003)LLS_ERROR_SERVICE_ALREADY_REGISTERED

API Call

LLS_Initialize_Service(BluetoothStackID, PASS_EventCallback, NULL, &PASSInstanceID)

API Prototype

int BTPSAPI LLS_Initialize_Service(unsigned int BluetoothStackID, LLS_Event_Callback_t EventCallback, unsigned long CallbackParameter, unsigned int *ServiceID)

Description of API

The following function is responsible for opening a LLS Server. The first parameter is the Bluetooth Stack ID on which to open the Server. The second parameter is the Callback function to call when an event occurs on this Server Port. The third parameter is a user-defined callback parameter that will be passed to the callback function with each event. The final parameter is a pointer to store the GATT Service ID of the registered LLS service. This can be used to include the service registered by this call. This function returns the positive, non-zero, Instance ID or a negative error code.

UnRegisterLLS

Description

This function is responsible for unregistering the Link Loss Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully registered PASP Service

(-4) FUNCTION_ERROR

(-1000)LLS_ERROR_INVALID_PARAMETER

(-1004)LLS_ERROR_INVALID_INSTANCE_ID

API Call

LLS_Cleanup_Service(BluetoothStackID, LLSInstanceID)

API Prototype

int BTPSAPI LLS_Cleanup_Service(unsigned int BluetoothStackID, unsigned int InstanceID)

Description of API

The following function is responsible for closing a previously LLS Server. The first parameter is the Bluetooth Stack ID on which to close the Server. The second parameter is the InstanceID that was returned from a successful call to LLS_Initialize_Service(). This function returns a zero if successful or a negative return error code if an error occurs.

DiscoverLLS

Description

This function is responsible for performing a LLS Service Discovery Operation. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) GDIS_Service_Discovery_Start success

(-4) Function_Error

(-1000)GDIS_ERROR_INVALID_PARAMETER

(-1001)GDIS_ERROR_NOT_INITIALIZED

(-1002)GDIS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003)GDIS_ERROR_INSUFFICIENT_RESOURCES

(-1009)GDIS_ERROR_SERVICE_DISCOVERY_OUTSTANDING

API Call

GATT_Start_Service_Discovery(BluetoothStackID, ConnectionID, (sizeof(UUID)/sizeof(GATT_UUID_t)), UUID, GATT_Service_Discovery_Event_Callback, sdLLS)

API Prototype

int BTPSAPI GDIS_Service_Discovery_Start(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GDIS_Event_Callback_t ServiceDiscoveryCallback, unsigned long ServiceDiscoveryCallbackParameter)

Description of API

The following function is used to initiate the Service Discovery process or queue additional requests. The function takes as its first parameter the BluetoothStackID that is associated with the Bluetooth Device. The second parameter is the connection ID of the remote device that is to be searched. The third and fourth parameters specify an optional list of UUIDs to search for. The final two parameters define the Callback function and parameter to use when the service discovery is complete. The function returns zero on success and a negative return value if there was an error.

GetAlertLevel

Description

The following function is responsible for reading the Alert Level. It can be executed by a Server or as a Client with an open connection to a remote Server. If executed as a Client, a GATT read request will be generated, and the results will be returned as a response in the GATT Client event callback. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Get Alert Level Request sent

(-4) FUNCTION_ERROR

(-1001) LLS_ERROR_INVALID_BLUETOOTH_STACK_ID (-1004) LLS_ERROR_INVALID_INSTANCE_ID

API Call

GATT_Read_Value_Request(BluetoothStackID, ConnectionID, DeviceInfo->LLS_ClientInfo.Alert_Level, GATT_ClientEventCallback_LLS, DeviceInfo->LLS_ClientInfo.Alert_Level)

API Prototype

int BTPSAPI GATT_Read_Value_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, GATT_Client_Event_Callback_t ClientEventCallback, unsigned long CallbackParameter)

Description of API

The following function is provided to allow a means of performing a read request on a remote device for a specific attribute value. The first parameter is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the attribute handle to read the value from. The final two parameters specify the GATT Client event callback function and callback parameter (respectively) that will be called when a response is received from the remote device. This function will return the positive, non-zero, Transaction ID of the request or a negative error code.

Note

If successful the return value will contain the Transaction ID that can be used to cancel the request.

SetAlertLevel

Description

The following function is responsible for writing the Alert Level. It can be executed by a Server or as a Client with an open connection to a remote Server if Client write access is supported. If executed as a Client, a GATT write request will be generated and the results will be returned as a response in the GATT Client event callback. This function will return zero on successful execution and a negative value on errors.

Parameters

SetAlertLevel has only one parameter which is an integer value that represents a Alert Level. This value is 0 = No Alert, 1 = Mild Alert, 2 = High Alert

Possible Return Values

(0) Ringer Setting successfully set

(-4) FUNCTION_ERROR

(-1000)LLS_ERROR_INVALID_PARAMETER

(-1001) LLS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1004)LLS_ERROR_INVALID_INSTANCE_ID

API Call

GATT_Write_Request(BluetoothStackID, ConnectionID, DeviceInfo->LLS_ClientInfo.Alert_Level, sizeof(NonAlignedByte_t), (void *)&Data, GATT_ClientEventCallback_LLS, DeviceInfo->LLS_ClientInfo.Alert_Level)

API Prototype

int BTPSAPI GATT_Write_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue, GATT_Client_Event_Callback_t ClientEventCallback, unsigned long CallbackParameter)

Description of API

The following function is provided to allow a means of performing a write request to a remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write the value of, followed by the length of the value (in bytes), followed by the the actual value data to write. The final two parameters specify the GATT Client event callback function and callback parameter (respectively) that will be called when a response is received from the remote device. This function will return the positive, non-zero, Transaction ID of the request or a negative error code.

Note

This function will not write an attribute value with a length greater than the current MTU - 3. To write a longer attribute value use the GATT_Prepare_Write_Request() function instead. This function may write less than the number of requested bytes. This can happen if the number of bytes to write is less than what can fit in the MTU for the specified connection. The data in the etGATT_Client_Write_Response, that is dispatched if the remote device accepts the request, indicates the number of bytes that were written to the remote device. If successful the return value will contain the Transaction ID that can be used to cancel the request.

RegisterIAS

Description

This function is responsible for registering the Immediate Alert Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

0) Successfully registered PASP Service

(-4) FUNCTION_ERROR

(-1000)IAS_ERROR_INVALID_PARAMETER

(-1002)IAS_ERROR_INSUFFICIENT_RESOURCES

(-1003)IAS_ERROR_SERVICE_ALREADY_REGISTERED

API Call

IAS_Initialize_Service(BluetoothStackID,IAS_EventCallback,0,&IASInstanceID)

API Prototype

int BTPSAPI IAS_Initialize_Service(unsigned int BluetoothStackID, IAS_Event_Callback_t EventCallback, unsigned long CallbackParameter, unsigned int *ServiceID)

Description of API

The following function is responsible for opening a IAS Server. The first parameter is the Bluetooth Stack ID on which to open the Server. The second parameter is the Callback function to call when an event occurs on this Server Port. The third parameter is a user-defined callback parameter that will be passed to the callback function with each event. The final parameter is a pointer to store the GATT Service ID of the registered IAS service. This can be used to include the service registered by this call. This function returns the positive, non-zero, Instance ID or a negative error code.

Note

Only 1 IAS may be open at a time, per Bluetooth Stack ID. All Client Requests will be dispatch to the EventCallback function that is specified by the second parameter to this function.

UnRegisterIAS

Description

This function is responsible for unregistering the Immediate Alert Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully registered PASP Service

(-4) FUNCTION_ERROR

(-1000)IAS_ERROR_INVALID_PARAMETER

(-1004)IAS_ERROR_INVALID_INSTANCE_ID

API Call

IAS_Cleanup_Service(BluetoothStackID, IASInstanceID)

API Prototype

int BTPSAPI IAS_Cleanup_Service(unsigned int BluetoothStackID, unsigned int InstanceID)

Description of API

The following function is responsible for closing a previously IAS Server. The first parameter is the Bluetooth Stack ID on which to close the Server. The second parameter is the InstanceID that was returned from a successful call to IAS_Initialize_Service(). This function returns a zero if successful or a negative return error code if an error occurs.

DiscoverIAS

Description

This function is responsible for performing an IAS Service Discovery Operation. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) GDIS_Service_Discovery_Start success

(-4) Function_Error

(-1000)GDIS_ERROR_INVALID_PARAMETER

(-1001)GDIS_ERROR_NOT_INITIALIZED

(-1002)GDIS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003)GDIS_ERROR_INSUFFICIENT_RESOURCES

(-1009)GDIS_ERROR_SERVICE_DISCOVERY_OUTSTANDING

API Call

GATT_Start_Service_Discovery(BluetoothStackID, ConnectionID, (sizeof(UUID)/sizeof(GATT_UUID_t)), UUID, GATT_Service_Discovery_Event_Callback, sdIAS)

API Prototype

int BTPSAPI GDIS_Service_Discovery_Start(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GDIS_Event_Callback_t ServiceDiscoveryCallback, unsigned long ServiceDiscoveryCallbackParameter)

Description of API

The following function is used to initiate the Service Discovery process or queue additional requests. The function takes as its first parameter the BluetoothStackID that is associated with the Bluetooth Device. The second parameter is the connection ID of the remote device that is to be searched. The third and fourth parameters specify an optional list of UUIDs to search for. The final two parameters define the Callback function

and parameter to use when the service discovery is complete. The function returns zero on success and a negative return value if there was an error.

ConfigureAlertLevel

Description

This function is responsible for performing an IAS Service Discovery Operation. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) GDIS_Service_Discovery_Start success

(-4) Function_Error

(-1000)GDIS_ERROR_INVALID_PARAMETER

(-1001)GDIS_ERROR_NOT_INITIALIZED

(-1002)GDIS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003)GDIS_ERROR_INSUFFICIENT_RESOURCES

(-1009)GDIS_ERROR_SERVICE_DISCOVERY_OUTSTANDING

API Call

GATT_Start_Service_Discovery(BluetoothStackID, ConnectionID, (sizeof(UUID)/sizeof(GATT_UUID_t)), UUID, GATT_Service_Discovery_Event_Callback, sdIAS)

API Prototype

int BTPSAPI GDIS_Service_Discovery_Start(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GDIS_Event_Callback_t ServiceDiscoveryCallback, unsigned long ServiceDiscoveryCallbackParameter)

Description of API

The following function is used to initiate the Service Discovery process or queue additional requests. The function takes as its first parameter the BluetoothStackID that is associated with the Bluetooth Device. The second parameter is the connection ID of the remote device that is to be searched. The third and fourth parameters specify an optional list of UUIDs to search for. The final two parameters define the Callback function and parameter to use when the service discovery is complete. The function returns zero on success and a negative return value if there was an error.

RegisterTPS

Description

This function is responsible for registering a TPS Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully registered PASP Service

(-4) FUNCTION_ERROR

(-1000)TPS_ERROR_INVALID_PARAMETER

(-1002)TPS_ERROR_INSUFFICIENT_RESOURCES

(-1003)TPS_ERROR_SERVICE_ALREADY_REGISTERED

API Call

TPS_Initialize_Service(BluetoothStackID, &TPSInstanceID)

API Prototype

int BTPSAPI TPS_Initialize_Service(unsigned int BluetoothStackID, unsigned int *ServiceID)

Description of API

The following function is responsible for opening a TPS Server. The first parameter is the Bluetooth Stack ID on which to open the Server. The second parameter is the Callback function to call when an event occurs on this Server Port. The third parameter is a user-defined callback parameter that will be passed to the callback function with each event. The final parameter is a pointer to store the GATT Service ID of the registered TPS service. This can be used to include the service registered by this call. This function returns the positive, non-zero, Instance ID or a negative error code.

UnRegisterTPS

Description

This function is responsible for unregistering a TPS Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully registered PASP Service

(-4) FUNCTION_ERROR

(-1000)LLS_ERROR_INVALID_PARAMETER

(-1004)LLS_ERROR_INVALID_INSTANCE_ID

API Call

TPS_Cleanup_Service(BluetoothStackID, TPSInstanceID)

API Prototype

int BTPSAPI TPS_Cleanup_Service(unsigned int BluetoothStackID, unsigned int InstanceID)

Description of API

The following function is responsible for closing a previously TPS Server. The first parameter is the Bluetooth Stack ID on which to close the Server. The second parameter is the InstanceID that was returned from a successful call to TPS_Initialize_Service(). This function returns a zero if successful or a negative return error code if an error occurs.

DiscoverTPS

Description

This function is responsible for performing a TPS Service Discovery Operation. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) GDIS_Service_Discovery_Start success

(-4) Function_Error

(-1000)GDIS_ERROR_INVALID_PARAMETER

(-1001)GDIS_ERROR_NOT_INITIALIZED

(-1002)GDIS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003)GDIS_ERROR_INSUFFICIENT_RESOURCES

(-1009)GDIS_ERROR_SERVICE_DISCOVERY_OUTSTANDING

API Call

GATT_Start_Service_Discovery(BluetoothStackID, ConnectionID, (sizeof(UUID)/sizeof(GATT_UUID_t)), UUID, GATT_Service_Discovery_Event_Callback, sdTPS)

API Prototype

int BTPSAPI GDIS_Service_Discovery_Start(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GDIS_Event_Callback_t ServiceDiscoveryCallback, unsigned long ServiceDiscoveryCallbackParameter)

Description of API

The following function is used to initiate the Service Discovery process or queue additional requests. The function takes as its first parameter the BluetoothStackID that is associated with the Bluetooth Device. The second parameter is the connection ID of the remote device that is to be searched. The third and fourth parameters specify an optional list of UUIDs to search for. The final two parameters define the Callback function and parameter to use when the service discovery is complete. The function returns zero on success and a negative return value if there was an error.

SetTxPowerLevel

Description

The following function is responsible for writing the Tx Power Level. It can be executed only by a Server. This function will return zero on successful execution and a negative value on errors.

Parameters

SetAlertLevel has only one parameter which is an integer value that represents a Tx Power Level.

Possible Return Values

(0) Ringer Setting successfully set

(-4) FUNCTION_ERROR

(-1000)TPS_ERROR_INVALID_PARAMETER

(-1001) TPS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1004)TPS_ERROR_INVALID_INSTANCE_ID

API Call

TPS_Set_Tx_Power_Level(BluetoothStackID, TPSInstanceID, (SByte_t)TempParam->Params[0].intParam)

API Prototype

int BTPSAPI TPS_Set_Tx_Power_Level(unsigned int BluetoothStackID, unsigned int InstanceID, SByte_t Tx_Power_Level)

Description of API

The following function is responsible for setting the Tx Power Level on the specified TPS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to TPS_Initialize_Server(). The final parameter is the Tx Power Level to set for the

specified TPS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

GetTxPowerLevel

Description

The following command is responsible for reading the Tx Power Level. It can be executed by a Server or as a Client with an open connection to a remote Server. If executed as a Client, a GATT read request will be generated, and the results will be returned as a response in the GATT Client event callback. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Get Alert Level Request sent

(-4) FUNCTION_ERROR

(-1001) LLS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1004) LLS_ERROR_INVALID_INSTANCE_ID

API Call

GATT_Read_Value_Request(BluetoothStackID, ConnectionID, DeviceInfo->LLS_ClientInfo.Alert_Level, GATT_ClientEventCallback_LLS, DeviceInfo->LLS_ClientInfo.Alert_Level)

API Prototype

int BTPSAPI GATT_Read_Value_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, GATT_Client_Event_Callback_t ClientEventCallback, unsigned long CallbackParameter)

Description of API

The following function is provided to allow a means of performing a read request on a remote device for a specific attribute value. The first parameter is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the attribute handle to read the value from. The final two parameters specify the GATT Client event callback function and callback parameter (respectively) that will be called when a response is received from the remote device. This function will return the positive, non-zero, Transaction ID of the request or a negative error code.

Note

If successful the return value will contain the Transaction ID that can be used to cancel the request.

20 FMP Demo Guide

20.1 Demo Overview

The Find Me profile defines the behavior when a button is pressed on a device to cause an immediate alert on a peer device. This can be used to allow users to find devices that have been misplaced. There are two roles defined in this profile, Server and the Client.

This application allows the user to use a console to use Bluetooth Low Energy (BLE) to establish connection between two BLE devices, change phone alert status, change ringer status, put in and get out of silent mode.

Note

The same instructions can be used to run this demo on the MSP432 or STM32F4 Platforms.

It is recommended that the user visits the kit setup Getting Started Guide for MSP430, Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as MSP-EXP430F5438 USB - Serial Port (COM x), XDS110 Class Application/User UART (COM x) for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud (9600 for MSP430), 8, n, 1. Once connected, reset the device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands.

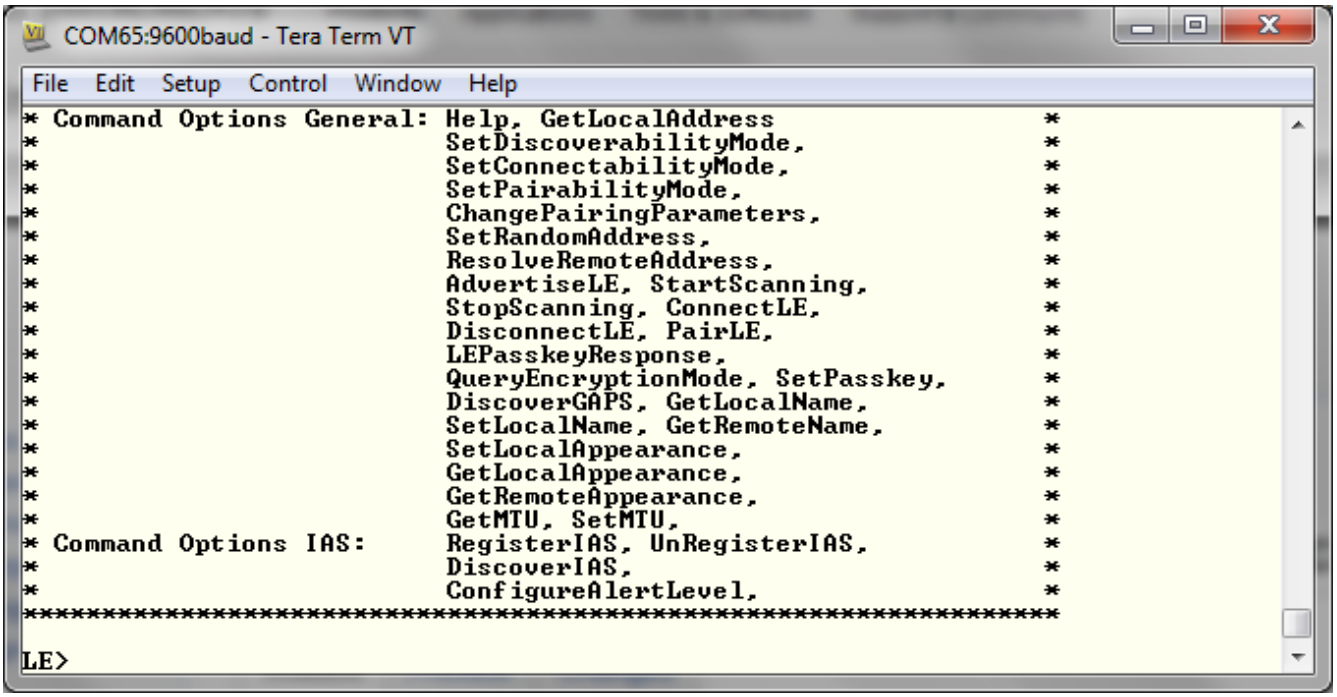

Figure 20-1. FMP Demo Start

20.2 Demo Application

The demo application provides a description on how to use the demo application to connect two configured boards and communicate over bluetoothLE. The included application registers a custom service on a board when the stack is initialized.

Device 1 (Server) setup on the demo application

1. To start, one of the devices has to have the Immediate Alert service running on it. It can be started by running RegisterIAS.

2. Next, the device acting as a Server needs to advertise to other devices. This can be done by running AdvertiseLE 1.

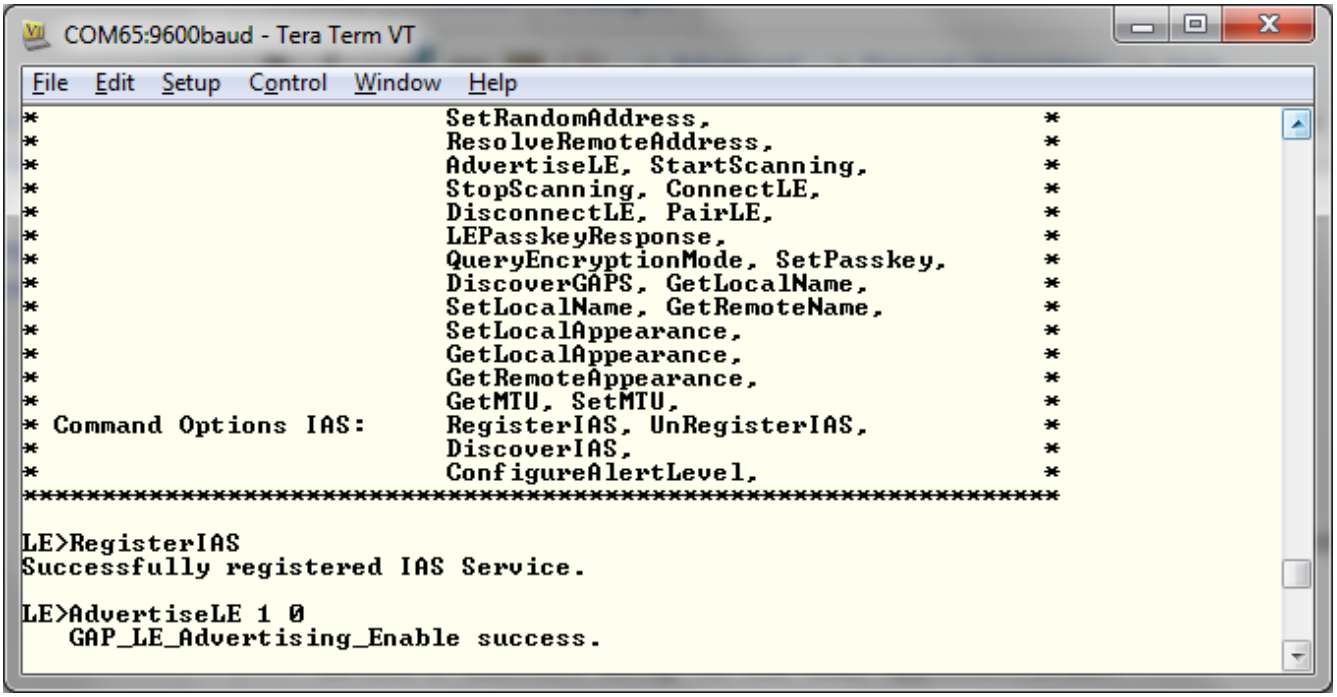

Figure 20-2. FMP Demo Register IAS Service

Device 2 (Client) setup on the demo application

> **Note**
> Steps 1 and 2 below are optional if you already know the Bluetooth address of the device that you want to connect to.

1. The Client LE device can try to find which LE devices are in the vicinity issuing the command: StartScanning.
2. Once you have found the device, you can stop scanning by issuing the command: StopScanning.

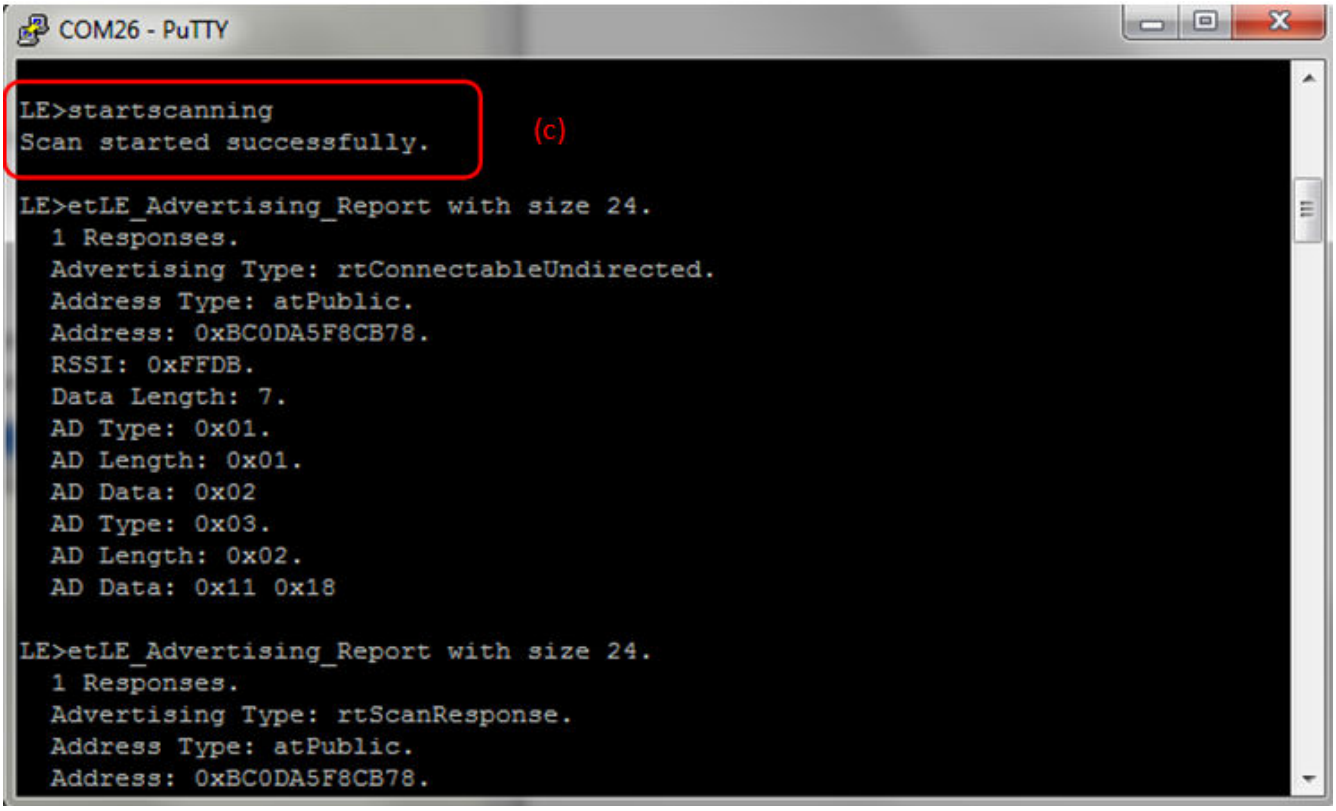

Figure 20-3. FMP Demo Scanning

Initiating connection from device 2

1. Once the application on the Client side knows the Bluetooth address of the device that is advertising, it can connect to that device issuing the command: ConnectLE <Bluetooth Address> 0 0

> **Note**
>
> 0 is used for both parameters since both devices use Public addresses, if using Random addresses use 1 for the address type parameters.

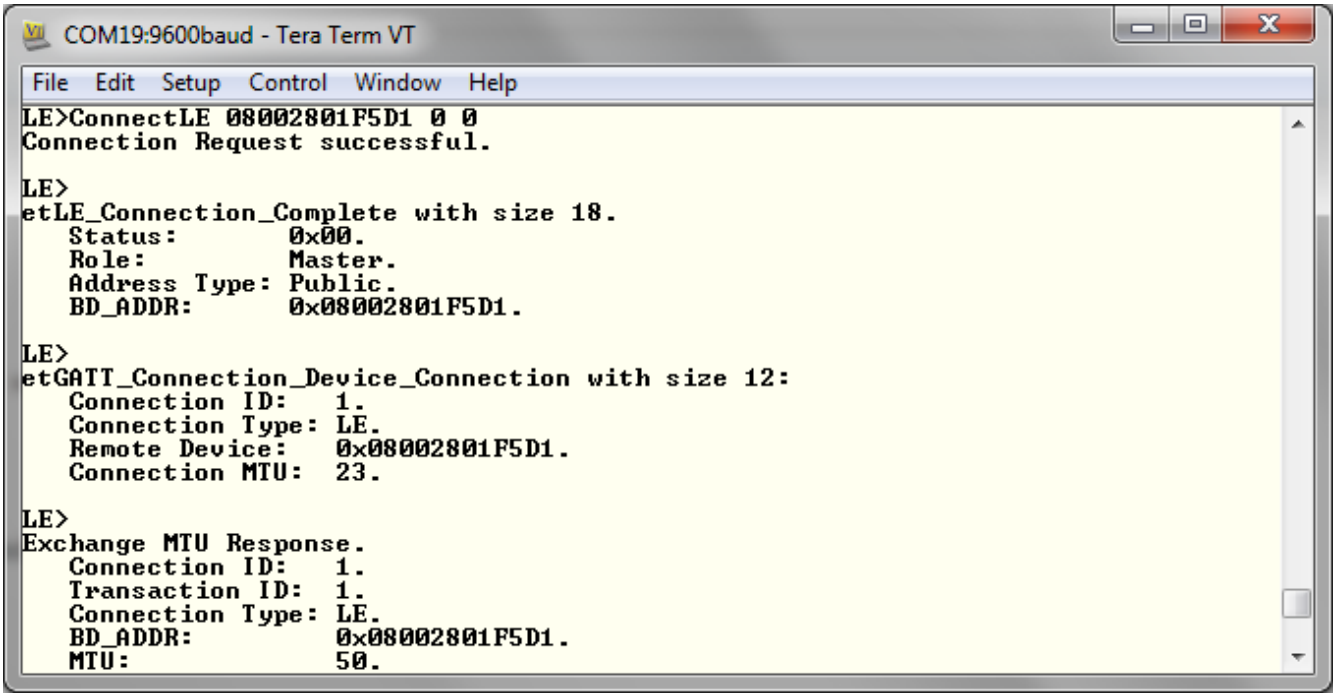

Figure 20-4. FMP Demo Connect LE

Identify supported services

1. After initialization, the Client needs to find out whether LLS, IAS and TPS services are supported. For this the commands, DiscoverLLS, DiscoverIAS, DiscoverTPS are run on the Client. After the service discovery operation is complete, the LLS, IAS and TPS Service Discovery Summary and list of supported features are shown.

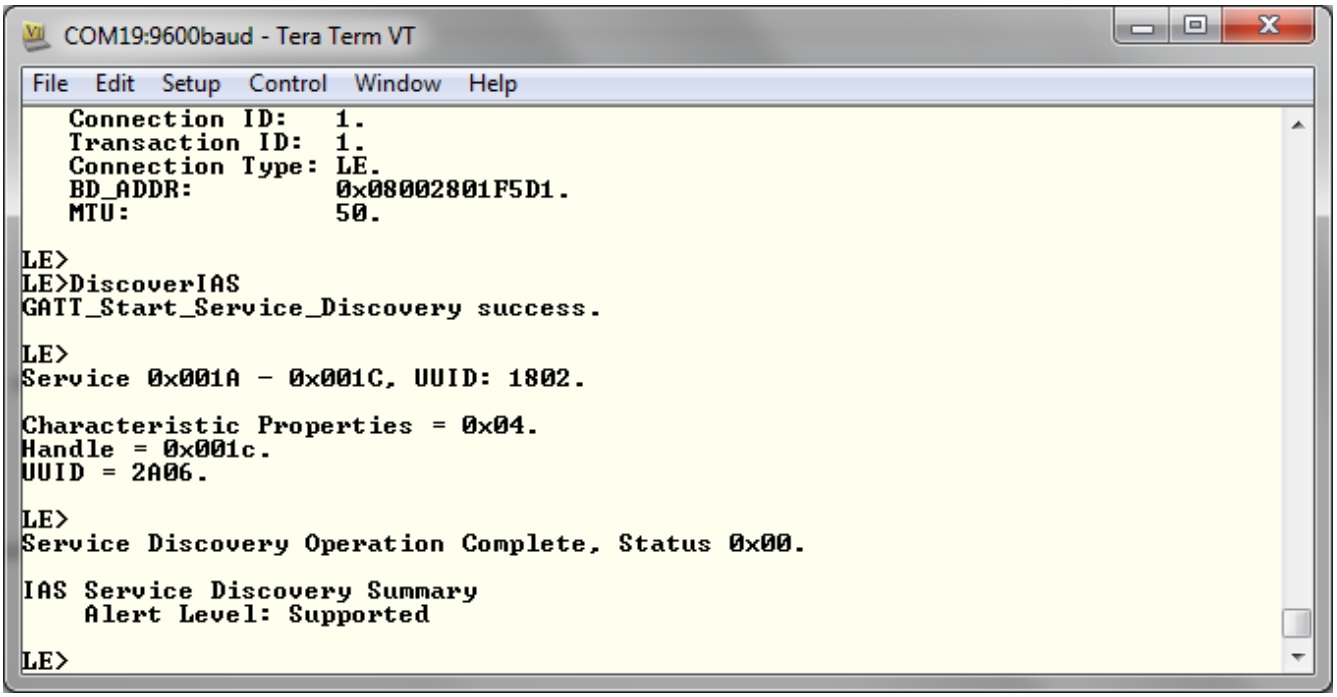

Figure 20-5. FMP Demo Discover IAS

Configuring Alert Level

1. The Client can configure the alert level which will get notified on the Server.

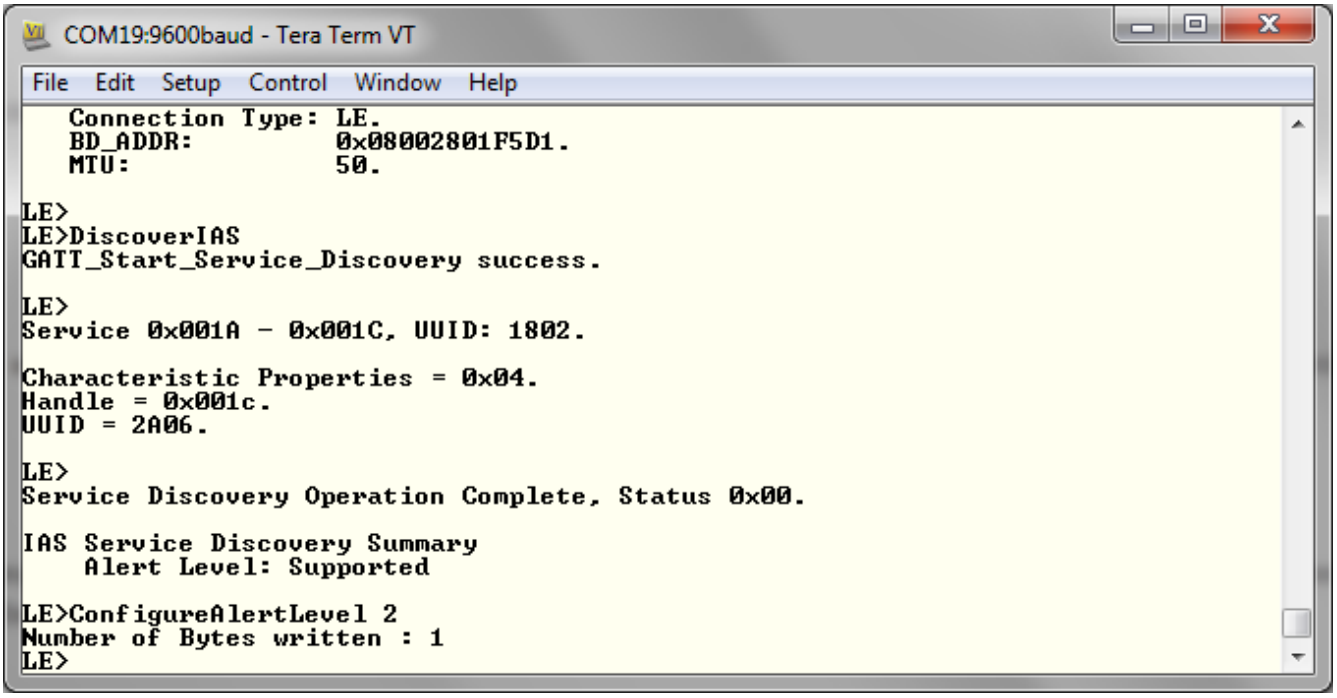

Figure 20-6. FMP Demo Discover IAS

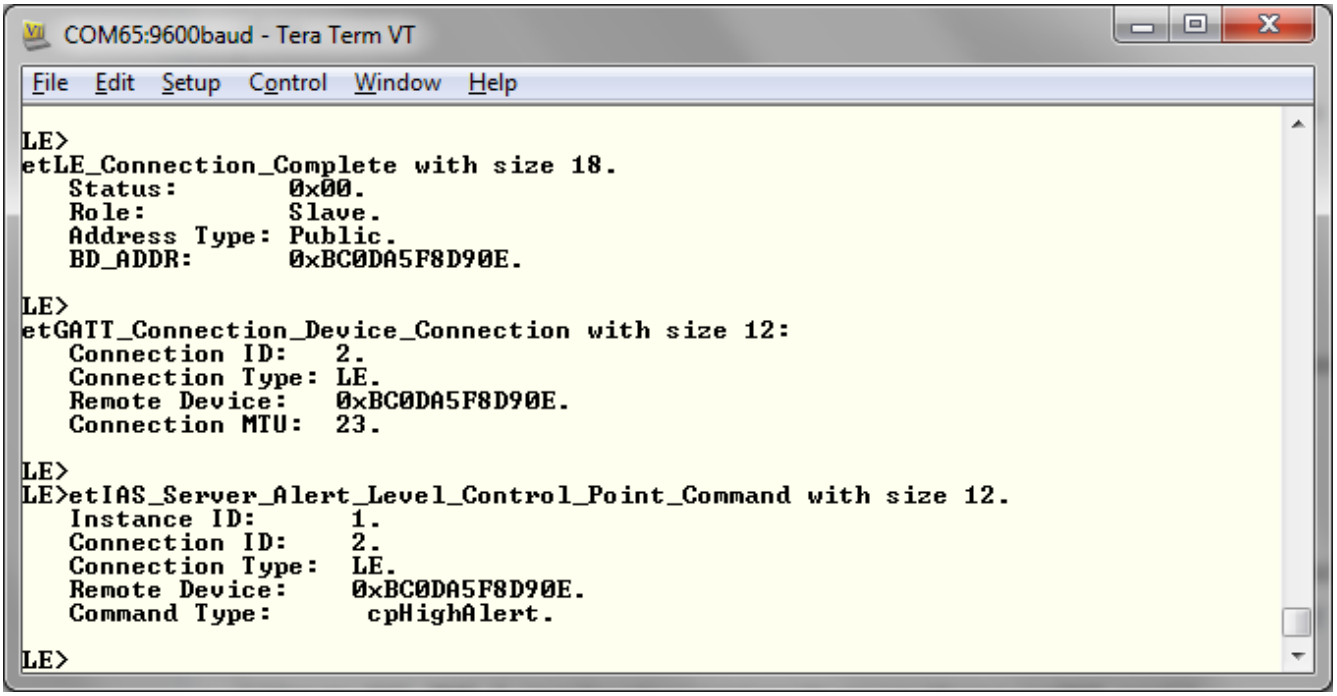

Figure 20-7. FMP Demo Connection Complete

20.3 Application Commands

RegisterIAS

Description

This function is responsible for registering a IAS Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

0) Successfully registered PASP Service

(-4) FUNCTION_ERROR

(-1000)IAS_ERROR_INVALID_PARAMETER

(-1002)IAS_ERROR_INSUFFICIENT_RESOURCES

(-1003)IAS_ERROR_SERVICE_ALREADY_REGISTERED

API Call

IAS_Initialize_Service(BluetoothStackID,IAS_EventCallback,0,&IASInstanceID)

API Prototype

int BTPSAPI IAS_Initialize_Service(unsigned int BluetoothStackID, IAS_Event_Callback_t EventCallback, unsigned long CallbackParameter, unsigned int *ServiceID)

Description of API

The following function is responsible for opening a IAS Server. The first parameter is the Bluetooth Stack ID on which to open the Server. The second parameter is the Callback function to call when an event occurs on this Server Port. The third parameter is a user-defined callback parameter that will be passed to the callback function with each event. The final parameter is a pointer to store the GATT Service ID of the registered IAS service. This can be used to include the service registered by this call. This function returns the positive, non-zero, Instance ID or a negative error code.

Note

Only 1 IAS may be open at a time, per Bluetooth Stack ID. All Client Requests will be dispatch to the EventCallback function that is specified by the second parameter to this function.

UnRegisterIAS

Description

This function is responsible for unregistering a IAS Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) Successfully registered PASP Service

(-4) FUNCTION_ERROR

(-1000)IAS_ERROR_INVALID_PARAMETER

(-1004)IAS_ERROR_INVALID_INSTANCE_ID

API Call

IAS_Cleanup_Service(BluetoothStackID, IASInstanceID)

API Prototype

int BTPSAPI IAS_Cleanup_Service(unsigned int BluetoothStackID, unsigned int InstanceID)

Description of API

The following function is responsible for closing a previously IAS Server. The first parameter is the Bluetooth Stack ID on which to close the Server. The second parameter is the InstanceID that was returned from a successful call to IAS_Initialize_Service(). This function returns a zero if successful or a negative return error code if an error occurs.

DiscoverIAS

Description

This function is responsible for performing a IAS Service Discovery Operation. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome of the command.

Possible Return Values

(0) GDIS_Service_Discovery_Start success

(-4) Function_Error

(-1000)GDIS_ERROR_INVALID_PARAMETER

(-1001)GDIS_ERROR_NOT_INITIALIZED

(-1002)GDIS_ERROR_INVALID_BLUETOOTH_STACK_ID

(-1003)GDIS_ERROR_INSUFFICIENT_RESOURCES

(-1009)GDIS_ERROR_SERVICE_DISCOVERY_OUTSTANDING

API Call

GATT_Start_Service_Discovery(BluetoothStackID, ConnectionID, (sizeof(UUID)/sizeof(GATT_UUID_t)), UUID, GATT_Service_Discovery_Event_Callback, sdIAS)

API Prototype

int BTPSAPI GDIS_Service_Discovery_Start(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GDIS_Event_Callback_t ServiceDiscoveryCallback, unsigned long ServiceDiscoveryCallbackParameter)

Description of API

The following function is used to initiate the Service Discovery process or queue additional requests. The function takes as its first parameter the BluetoothStackID that is associated with the Bluetooth Device. The second parameter is the connection ID of the remote device that is to be searched. The third and fourth parameters specify an optional list of UUIDs to search for. The final two parameters define the Callback function and parameter to use when the service discovery is complete. The function returns zero on success and a negative return value if there was an error.

ConfigureAlertLevel

Description

The following function is responsible for writing the Alert Level Control Point to connected remote device. It can be executed only by a Client. This function will return zero on successful execution and a negative value on errors.

Parameters

SetRingerSetting has only one parameter which is an integer value that represents the Alert Level. This value is 0 = cpNoAlert, 1 = cpMildAlert, 2 = cpHighAlert.

Possible Return Values

(0) Ringer Setting successfully set

(-4) FUNCTION_ERROR

(-1000)IAS_ERROR_INVALID_PARAMETER

(-1003)IAS_ERROR_SERVICE_ALREADY_REGISTERED

(-1004)IAS_ERROR_INVALID_INSTANCE_ID

(-1005)IAS_ERROR_MALFORMATTED_DATA

API Call

GATT_Write_Without_Response_Request(BluetoothStackID, ConnectionID, DeviceInfo->IAS_ClientInfo.Control_Point, IAS_ALERT_LEVEL_CONTROL_POINT_VALUE_LENGTH, ((void *)Buffer))

API Prototype

int BTPSAPI GATT_Write_Without_Response_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue)

Description of API

The following function is provided to allow a means of performing a write without response request to remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write, followed by the length of the value data to write (in bytes), followed by the actual value to write. This function will return the number of bytes written on success or a negative error code.

21 CSCP Demo Guide

21.1 Demo Overview

Note

The same instructions can be used to run this demo on the MSP432 or STM32F4 Platforms.

The Cycling Speed and Cadence Profile (CSCP) enables users to track the speed and the cadence during a cycling exercise period. There are two roles defined in this profile. The first is the the Server, typically a bluetooth enabled sensor that measures the wheel or the cadence speed and rotations and transmits these measurements. The second is the Client which can be a device like a phone, smart watch, an auxillary display or even a tablet that gets a measurement everytime an event happens on the Server.

This application allows the user to use a console to use Bluetooth Low Energy (BLE) to establish connection between two BLE devices, notify measurements and change the sensor location and cumulative wheel value in the Speed Cadence (SC) Control Point.

It is recommended that the user visits the kit setup Getting Started Guide for MSP432 or Getting Started Guide for STM32F4 pages before trying the application described on this page.

Running the Bluetooth Code

Once the code is flashed, connect the board to a PC using a miniUSB or microUSB cable. Once connected, wait for the driver to install. It will show up as **XDS110 Class Application/User UART (COM x)** for MSP432, under Ports (COM & LPT) in the Device manager. Attach a Terminal program like PuTTY to the serial port x for the board. The serial parameters to use are 115200 Baud, 8, n, 1. Once connected, reset the device using Reset S3 button (located next to the mini USB connector for the MSP430) and you should see the stack getting initialized on the terminal and the help screen will be displayed, which shows all of the commands. This device will become the Server.

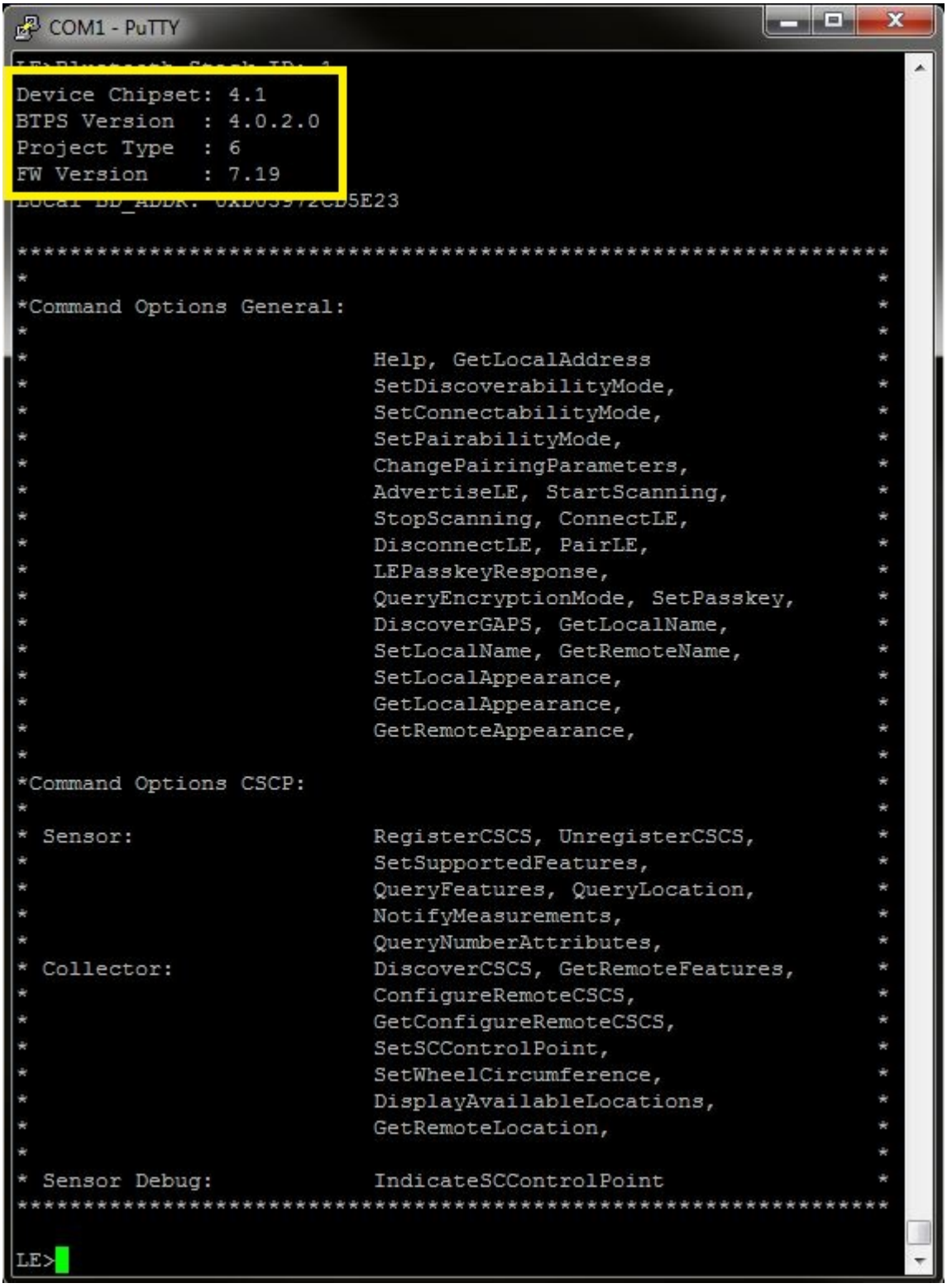

Figure 21-1. CSCP Demo PUTTY Start Screen

Note

The yellow square holds the FW and BTPS versions for future use. Now use the second board and follow the same steps performed before when running the Bluetooth code on the first board. The second device that is connected to the computer will be the Client.

21.2 Demo Application

This section provides a description of how to use the demo application to connect two configured boards and communicate over bluetooth. Bluetooth CSCP is a simple Client-Server connection process. We will setup one of the boards as a Server and the other board as a Client. We will then initiate a connection from the Client to the Server. Once connected, we can transmit data between the two devices over bluetooth.

Server setup on the demo application

1. We will setup the first board as a Server. Note the bluetooth address of the Server; we will later use this to initiate a connection from the Client.Up to four commands are all that is needed to setup the CSCS Server.
2. The first command we need is RegisterCSCS, this command will register the service and initiate the Server, issue the RegisterCSCS command.

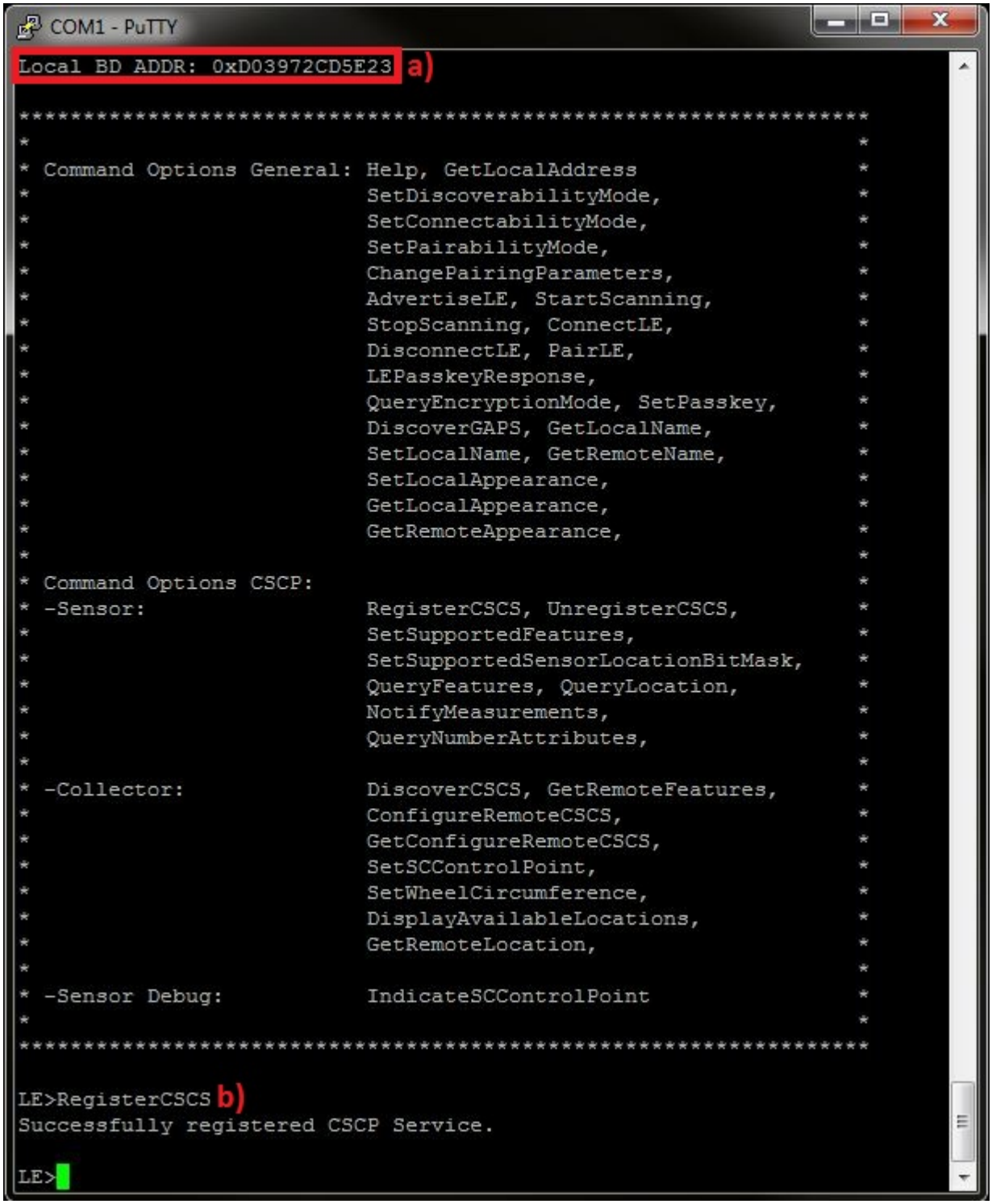

Figure 21-2. CSCP Demo CSCS Register

3. In order to set the supported features we issue the **SetSupportedFeatures <Wheel Measurement Support> <Crank Measurement Support> <Multiple Location Support>**.

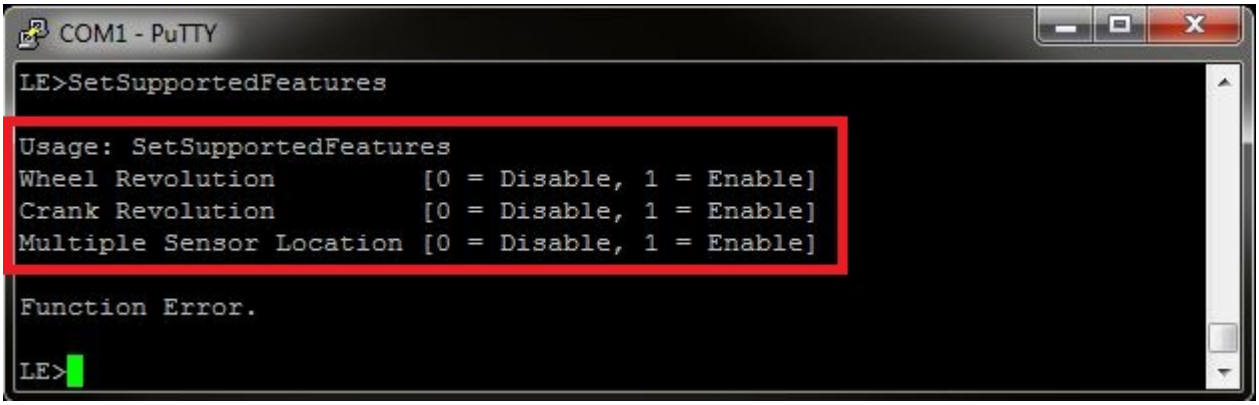

Figure 21-3. CSCP Demo STM32 Set Supported Features Usage

Issue the **SetSupportedFeatures 1 1 1** command to enable all features, the terminal will display a note that indicates the we need to issue the **GetRemoteFeatures** command in the Client's terminal.

4. If Multiple Location is supported, in order to set the supported sensor locations list we issue the **SetSupportedSensorLocationBitMask <BitMask>** command, which is a bitmask that will enable the supported locations in the Server.

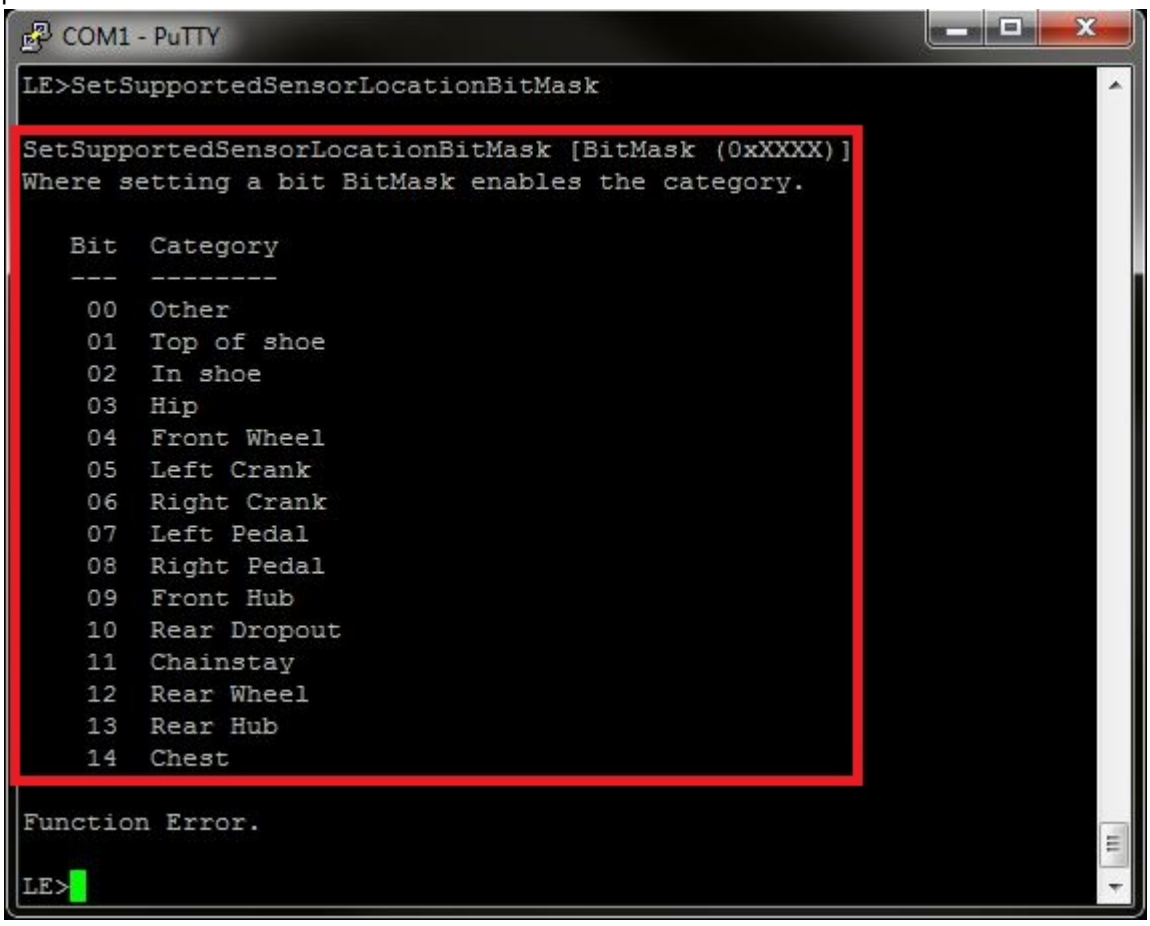

Figure 21-4. CSCP Demo CSCS Set Supported Sensor Location Bit Mask Usage

Issue the **SetSupportedSensorLocationBitMask 0x7fff** command to set all the available locations in the list, the terminal will display that the update was successful.

5. Now, after we configured the Server, we can begin advertising to nearby devices by issuing the **AdvertiseLE 1** command.

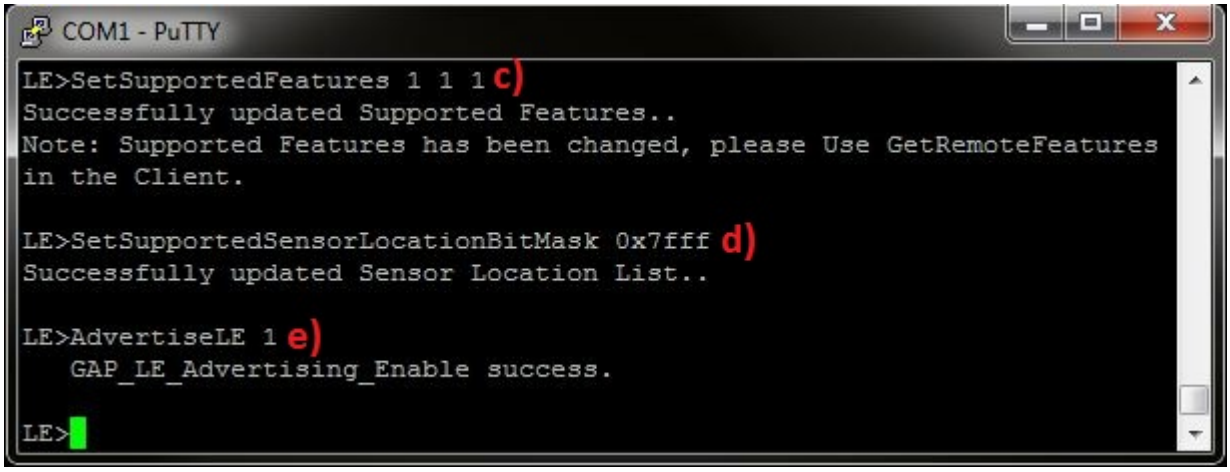

Figure 21-5. CSCP Demo CSCS Features and Location

Initiating connection from the Client

Note

Steps 1 and 2 are optional if you already know the Bluetooth address of the device that you want to connect to.

1. The Client LE device can try to find which LE devices are in the vicinity by issuing the **StartScanning** command.
2. Once you have found the device, you can stop scanning by issuing the **StopScanning** command.

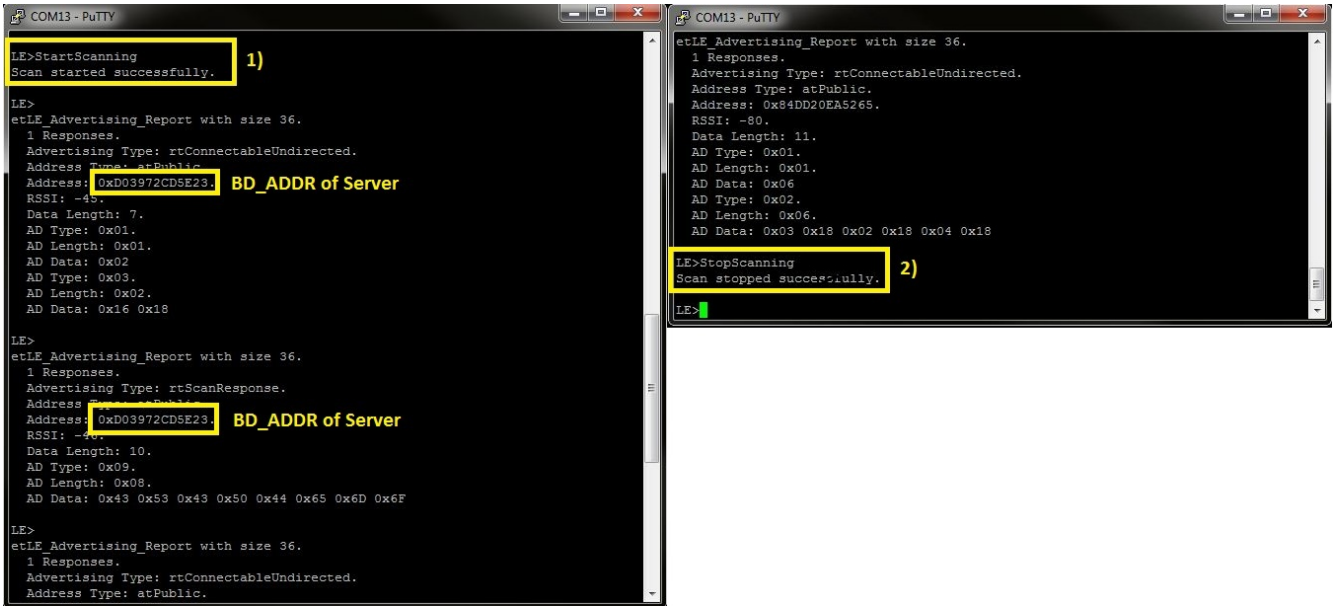

Figure 21-6. CSCP Demo STM32 Start Scanning

3. Retrieve the Bluetooth address of the first board that was configured as a Server.
4. Issue a **ConnectLE <BD_ADDR of Server>** command in the Client terminal.
5. When a Client successfully connects to a Server, both the Client and Server will output **LE_Connection_Complete** notification and information about the current connection.

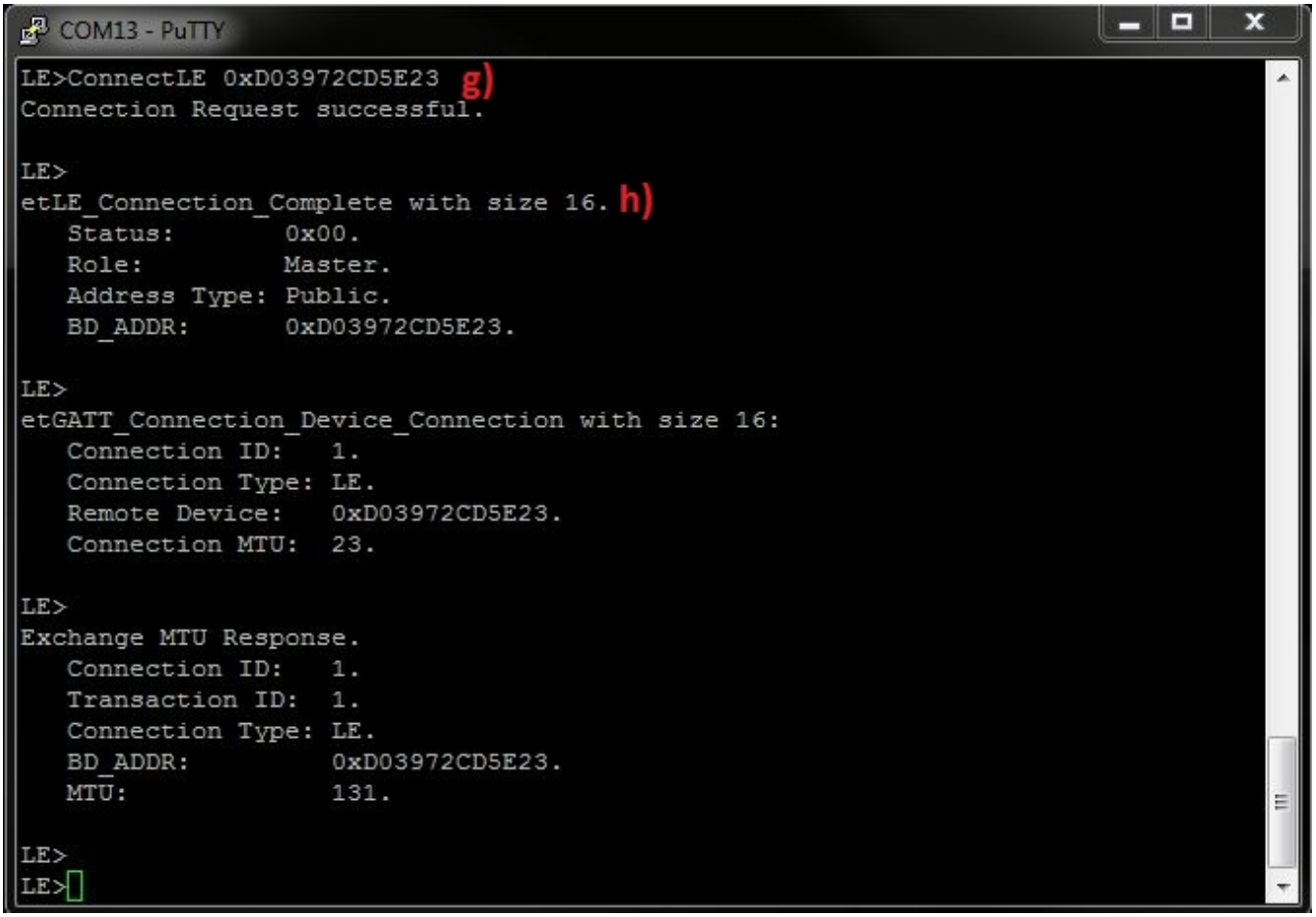

Figure 21-7. CSCP Demo STM32 Client Connect LE

Discovering and Configuring the Server

Now we have a connection established and both devices are ready to send data to each other. Before Cycle Speed and Cadence information can be sent we must first initialize commands in the Client terminal.

6. In order to discover The service handles we issue the **DiscoverCSCS** command and the handles that are discovered will be displayed in the terminal.

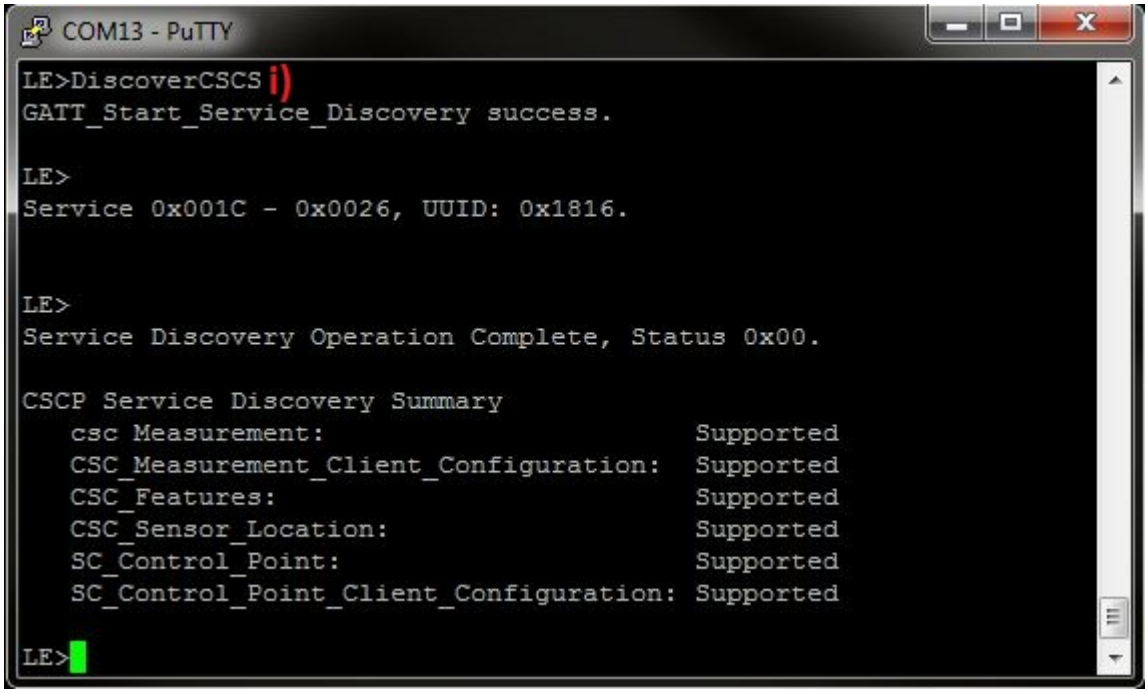

Figure 21-8. CSCP Demo STM32 Discover CSCS

7. After discovering the handles we need to discover the supported features of the Server, so we issue the GetRemoteFeatures command, which will display the supported features in the Client terminal.

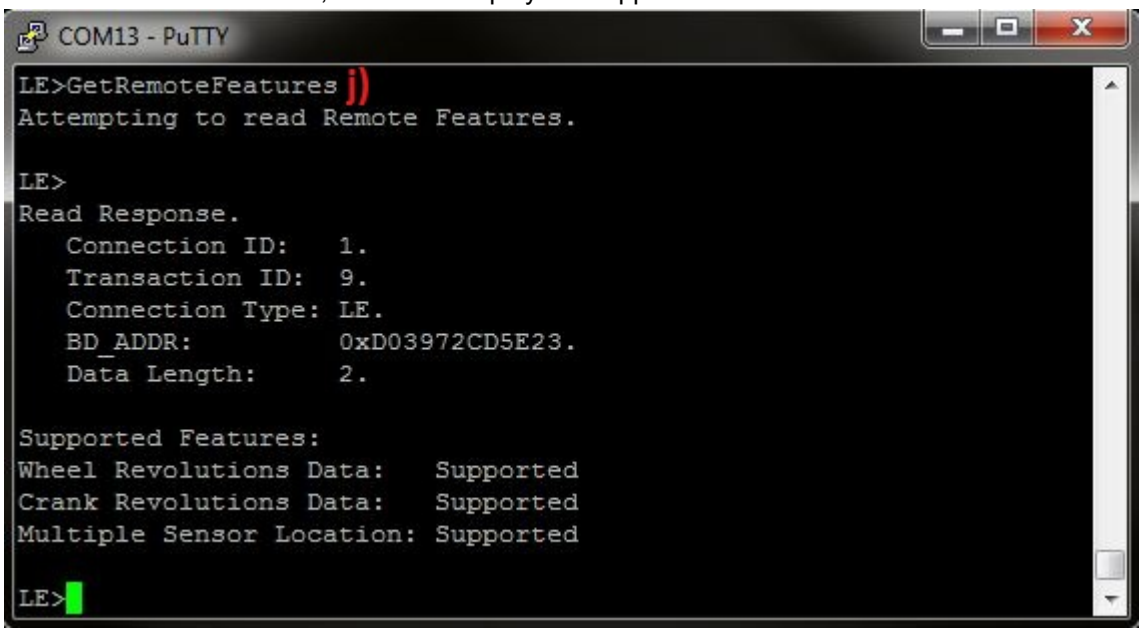

Figure 21-9. CSCP Demo STM32 Get Remote Features

8. In order to enable Server notifications, and if the SC Control Point exists also enable indications we issue the **ConfigureRemoteCSCS 1** command in the Client's terminal. If the configuration succeeds information about the CCCD configuration will be displayed in the Client and Server terminals.

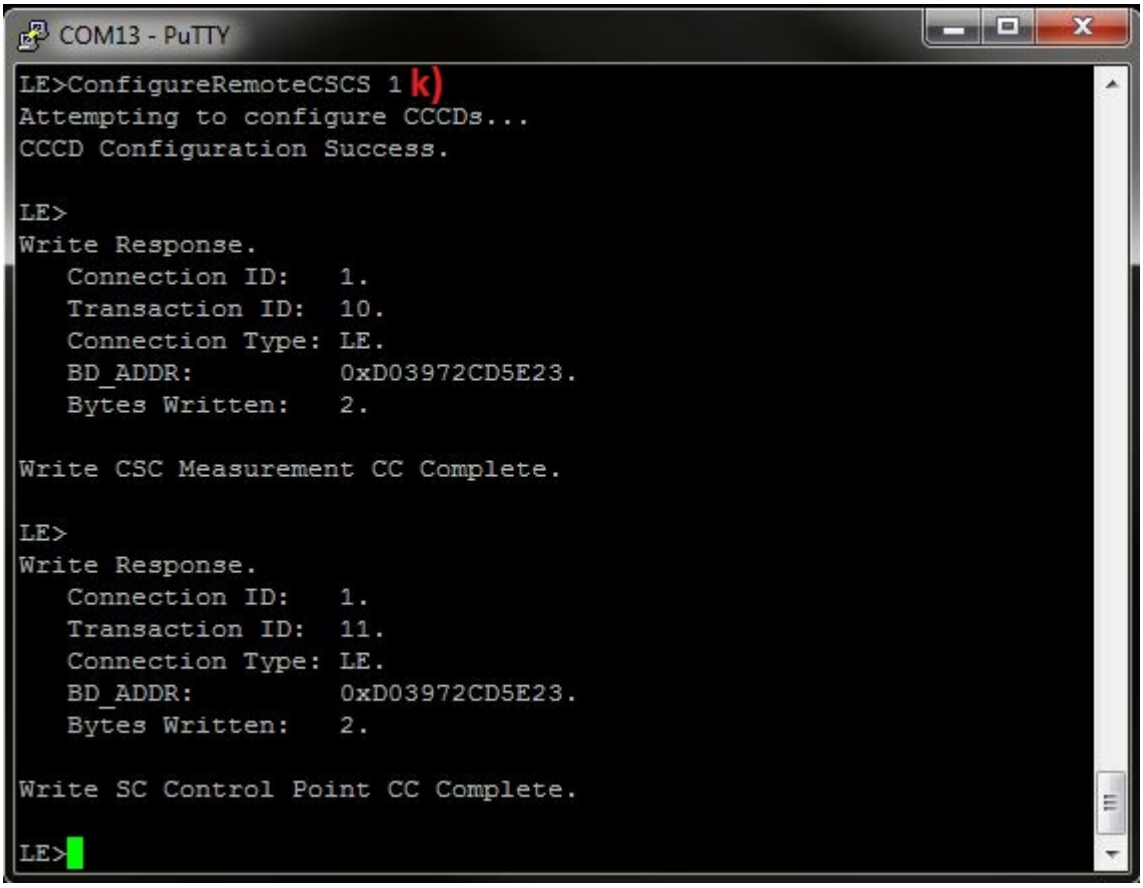

Figure 21-10. CSCP Demo STM32 Configure Remote Enable

Transferring Cycle Speed and Cadence Information

Now we have a connection established and both devices configured and are ready to send data to each other.

We can now notify from the Sensor (Jump to paragraph **t**) or we can send SC Control Point Command to the sensor from the collector (next paragraph).

In Client Terminal:

1. When we make new connection between collector and Sensor we need to Set the wheel circumference so the collector will be able to calculate Instantaneous speed, so we issue the **SetWheelCircumference <Circumference>** command to set the circumference in cm. Type **SetWheelCircumference 210** in order to set wheel circumference of **210 cm**.

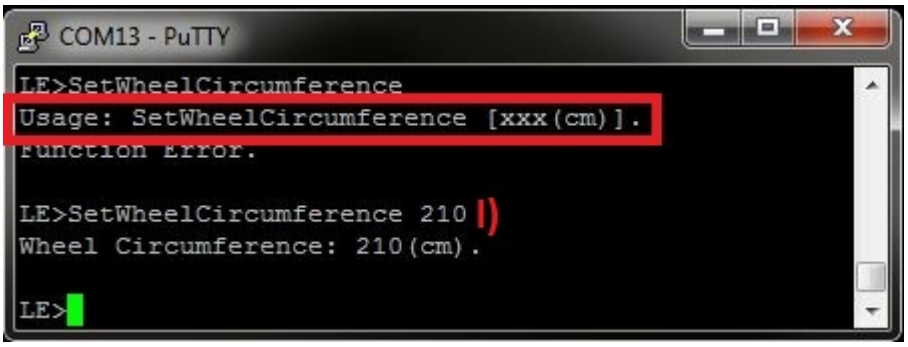

Figure 21-11. CSCP Demo STM32 Set Wheel Circumference

2. When we want to change data in the sensor we can do it only from the collector by issuing the **SetSCControlPoint** command. We can update the cumulative wheel revolutions, the sensor location or retrieve a list of supported sensor locations using this command. After each command the collector will receive a response indication from the sensor.

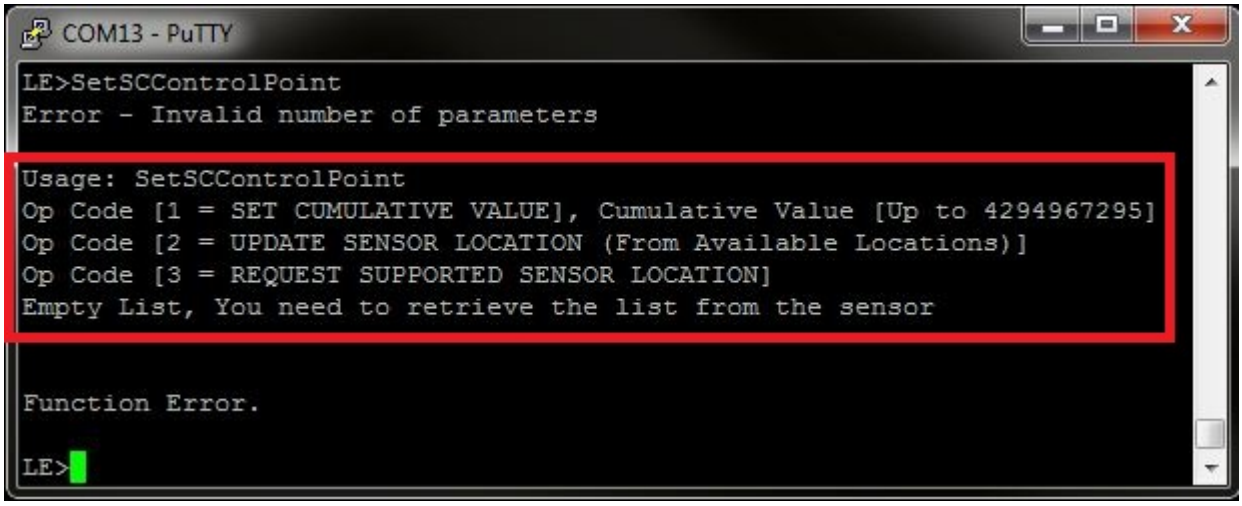

Figure 21-12. CSCP Demo STM32 Set SC Control Point Usage

3. If we want, and only when we support wheel revolutions, we can update the cumulative wheel revolutions using the SC Control Point Command. To do so we issue the **SetSCControlPoint 1 <Cumulative value to update>** command, for example type **SetSCControlPoint 1 2000**.

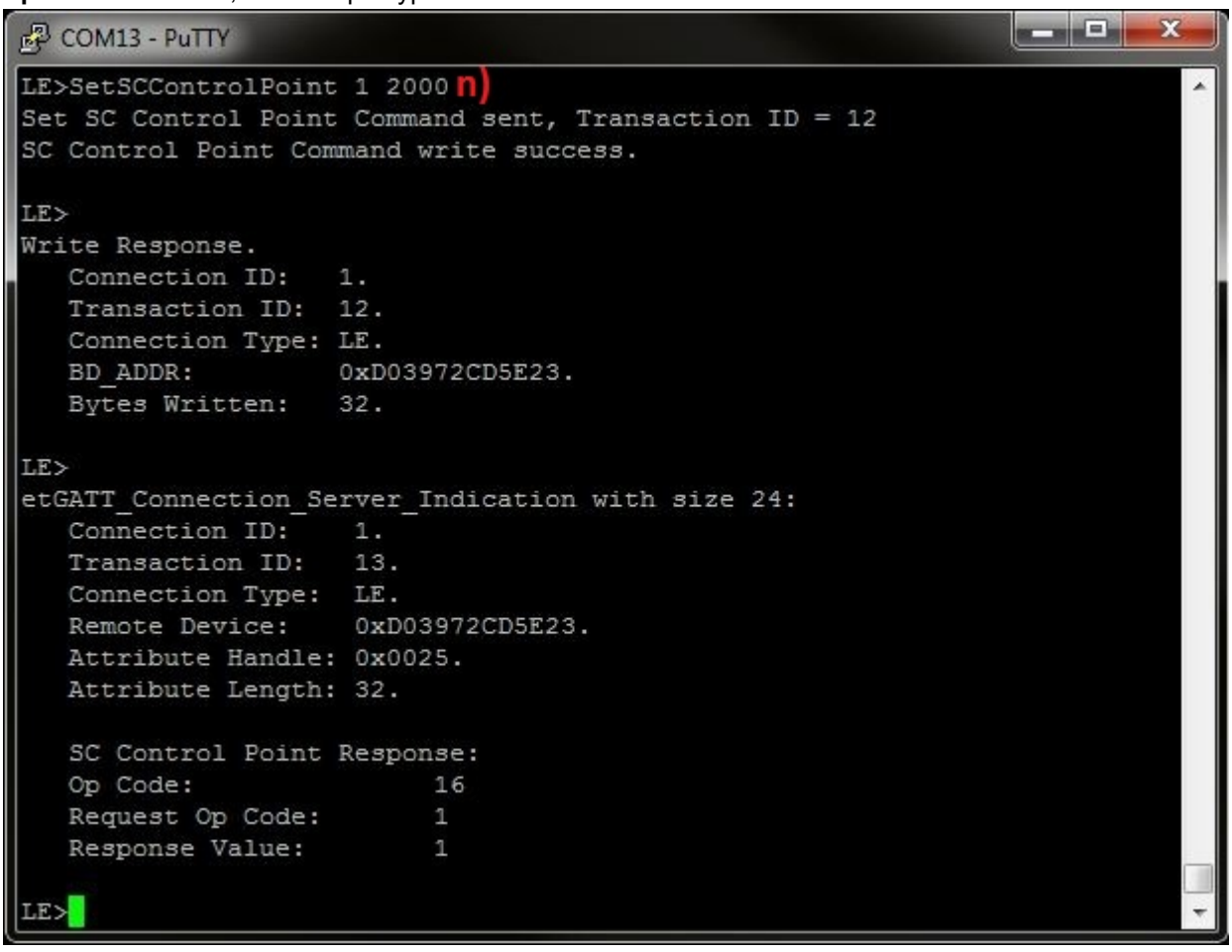

Figure 21-13. CSCP Demo Set SC Control Point

4. If we want to change sensor location, and only if we support multiple sensor location, we can change it only to locations that the sensor supports. To see the supported locations list, we issue the **DisplayAvailableLocations** command. If the list is empty we need to retrieve it first, we will do so in the next paragraph. If the list isn't empty jump to paragraph "p".

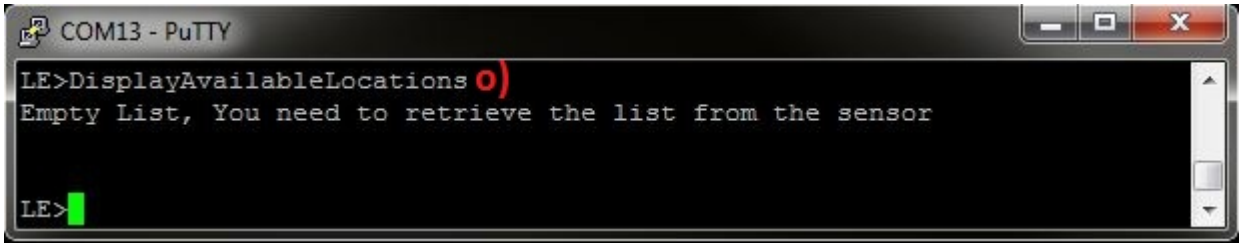

Figure 21-14. CSCP Demo STM32 Display Available Locations

5. In order to retrieve the sensor location list, and only when we supported multiple sensor location, we need to use the set SC Control Point command which retrieve the list, to do so we use SetSCControlPoint 3. Type SetSCControlPoint 3 and you will see the available sensor locations.
6. You can see confirmation for the command and the indication from the sensor that includes the list.

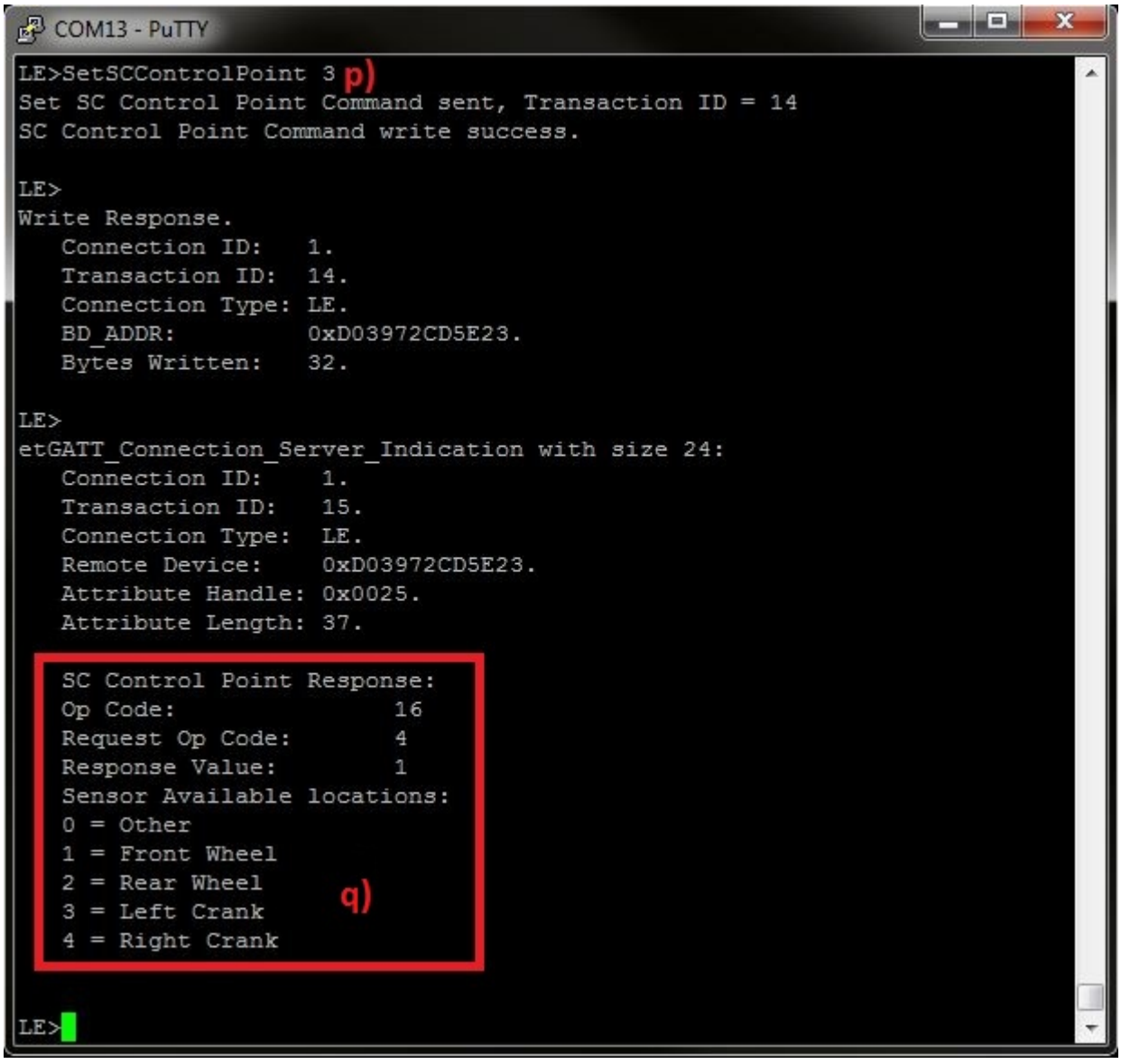

Figure 21-15. CSCP Demo STM32 Set SC Control Point 2

7. After the collector has the sensor supported list locations, the collector can update the sensor location using **SetSCControlPoint 2** and the number of the location from the list that we want to update, type **SetSCControlPoint 2 3**.

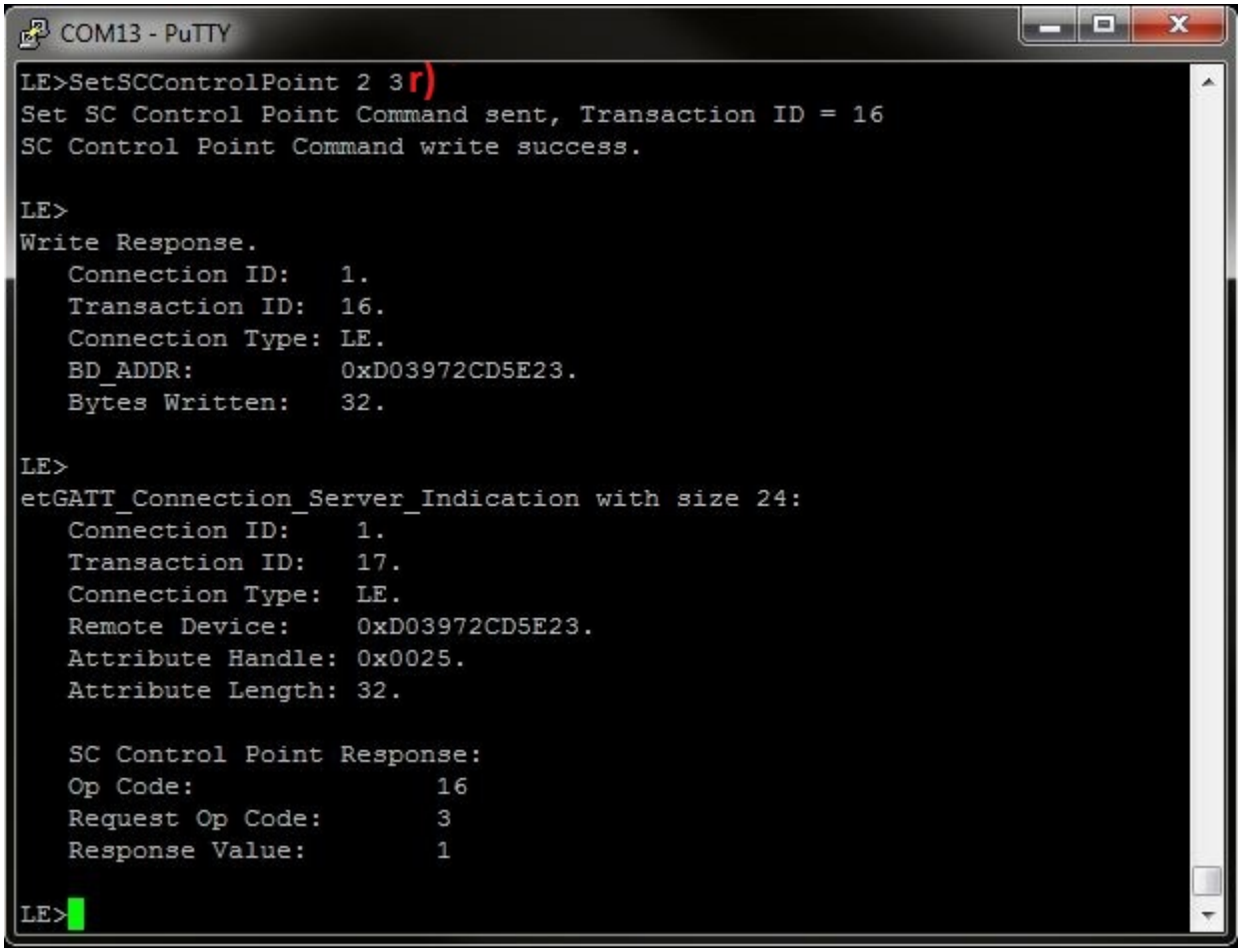

Figure 21-16. CSCP Demo STM32 Set CS Control Point 3

In Server Terminal:

8. After we set new sensor location, we can check it by issuing the Querylocation command and you will be able to see the new location.

Figure 21-17. CSCP Demo STM32 Query Location

9. Now, when we finish with all the configurations and command the sensor can start notifying his measurements, in order to do so, we issue the **NotifyMeasurements** command. When the sensor support only wheel revolutions or crank revolutions you need to input 2 parameters, cumulative value, and time event value, when both are supported you need to input 4 parameters wheel cumulative value, wheel time event value, crank cumulative value and crank time event value.

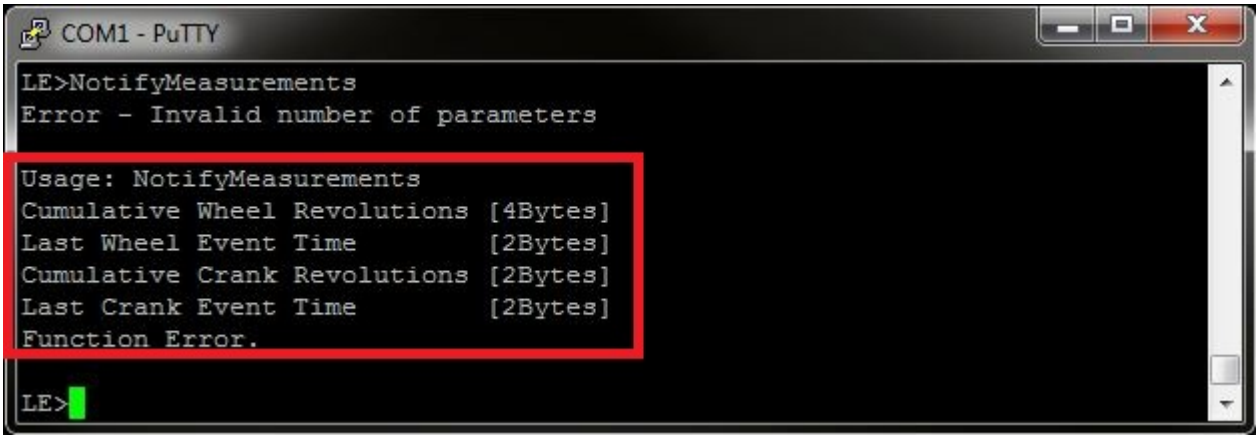

Figure 21-18. CSCP Demo STM32 Notify Usage

10. In this example wheel revolutions and crank revolutions are supported. then type **NotifyMeasurements 2008 64000 65534 9300**, you will be able to see on the Client terminal the parameters you entered.
    - Server Terminal

Figure 21-19. CSCP Demo STM32 Notify

- Client Terminal
- The Calculation for instantaneous speed

  Speed[cm/(s/1024)] = (Difference in two successive Cumulative Wheel Revolution values * Wheel Circumference) / (Difference in two successive Last Wheel Event Time values) In order to receive Speed[km/h] we need to multiple the result with (3.6*1024/100) - The Demo is doing the calculation for us when we use DISPLAY_DEBUG definitions in the environment
- The Calculation for instantaneous cadence

  Cadence[r/(s/1024)] = (Difference in two successive Cumulative Crank Revolution values) / (Difference in two successive Last Crank Event Time values) In order to receive Cadence[rpm] we need to multiple the result with (60*1024) - The Demo is doing the calculation for us when we use DISPLAY_DEBUG definitions in the environment

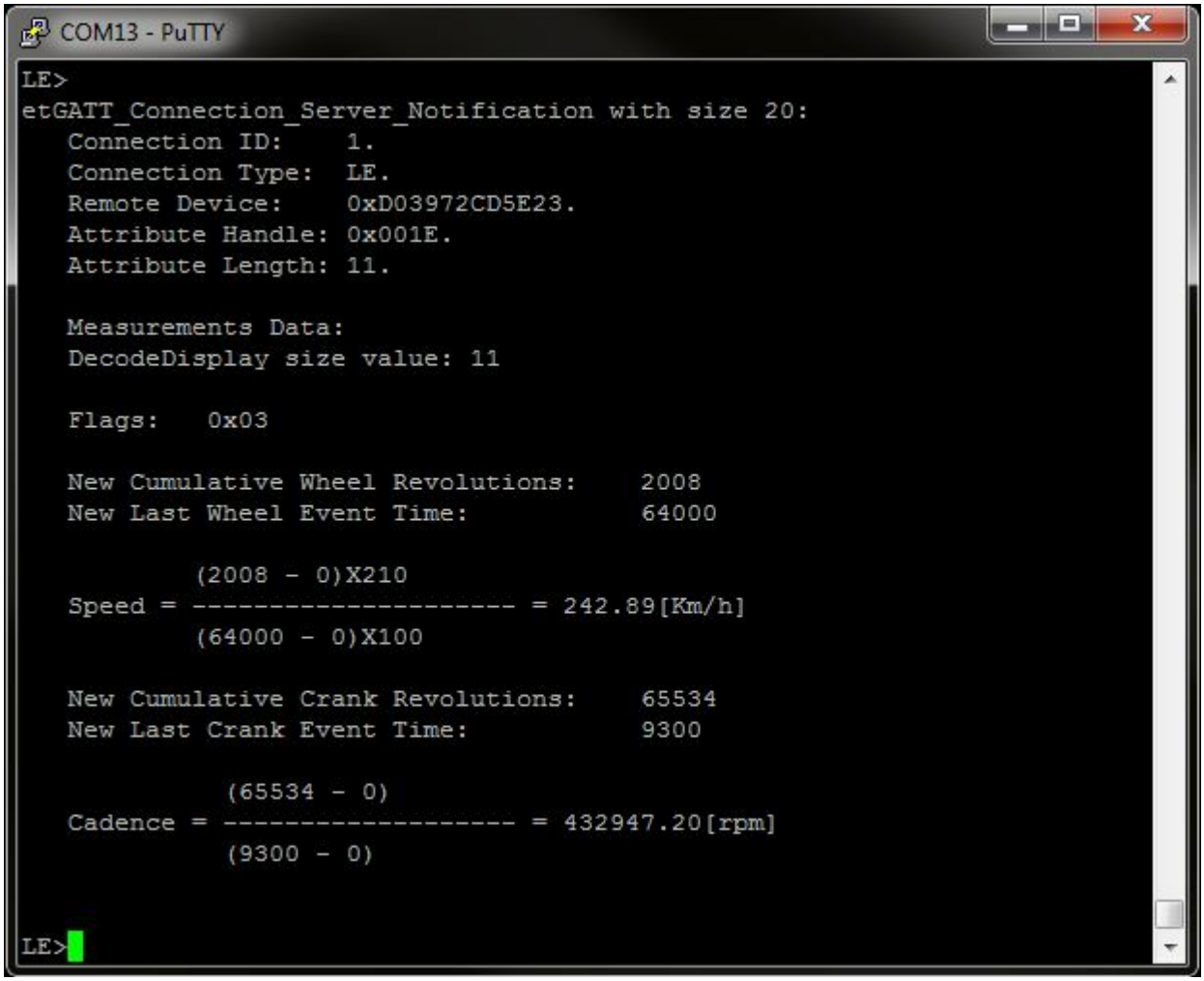

Figure 21-20. CSCP Demo STM32 Notify Client

11. When using debug mode, the application will calculate the Instantaneous Speed and Instantaneous Cadence. In order to do this calculation we need to send another notification. Type NotifyMeasurements 2016 65024 65535 10324, now in the Client Terminal you can see Instantaneous Speed [km/h] and Instantaneous Cadence [rpm].
    - Server Terminal

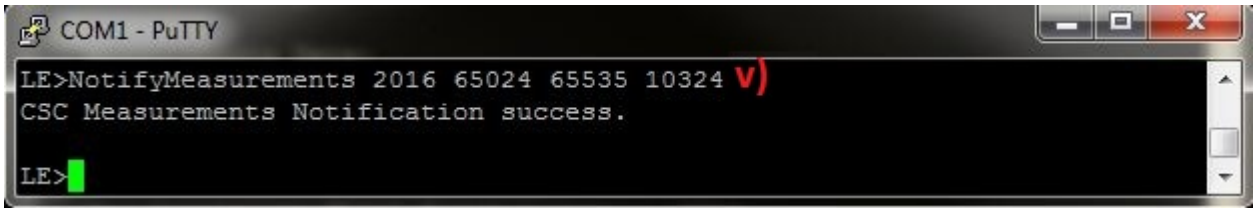

Figure 21-21. CSCP Demo STM32 Notify 2

- Client Terminal

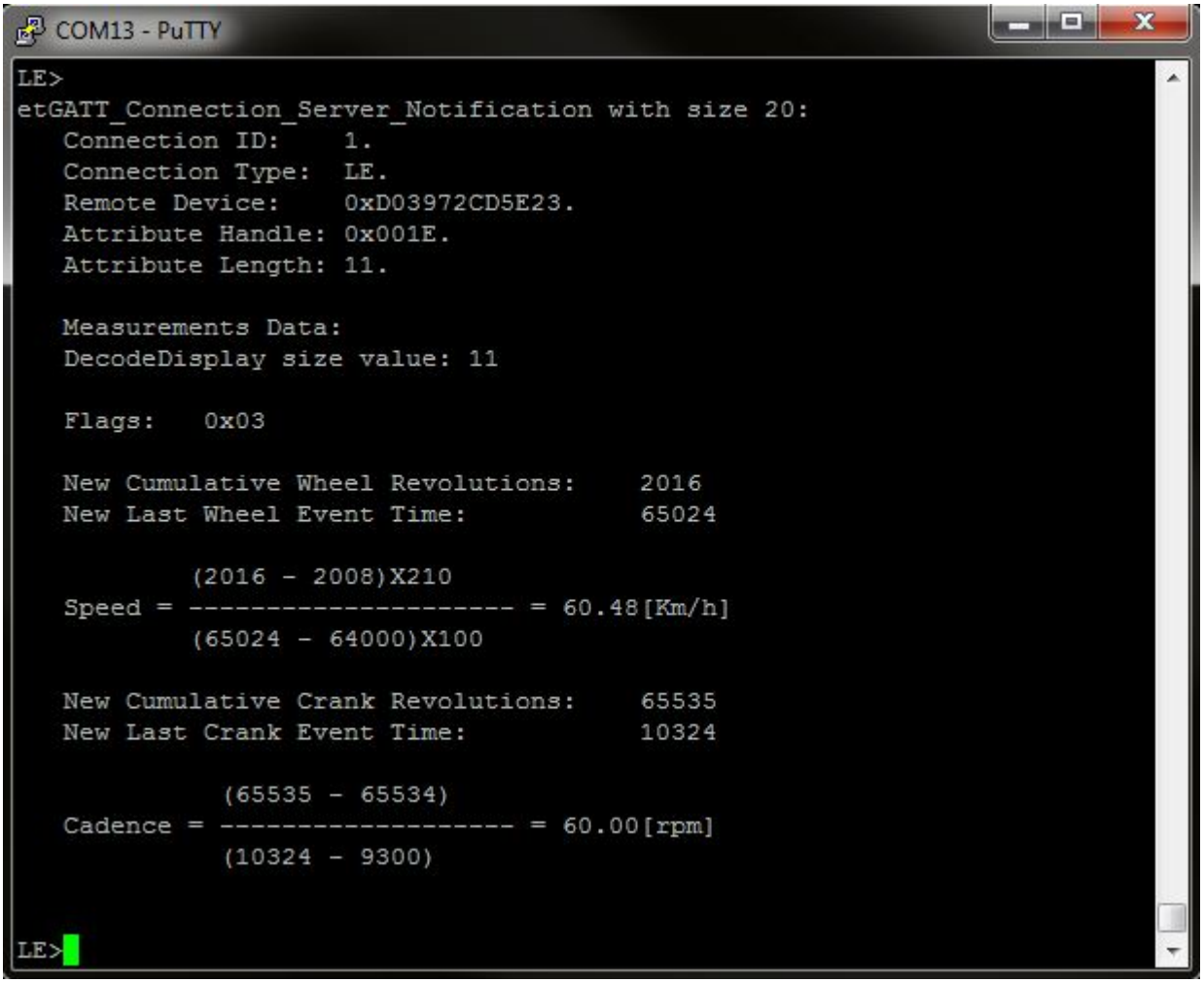

Figure 21-22. CSCP Demo STM32 Notify Client 2

21.3 Application Commands

TI's Bluetooth stack is an implementation of the upper layers of the Bluetooth protocol stack. TI's Bluetooth stack provides a robust and flexible software development tool that implements the Bluetooth Protocols and Profiles above the Host Controller Interface (HCI). TI's Bluetooth stack's Application Programming Interface (API) provides access to the upper-layer protocols and profiles and can interface directly with a Bluetooth controller chip.

The basic Bluetooth application included with STM3240G-eval is a Cycling Speed and Cadence Application.

An overview of the application and other applications can be read at the Getting Started Guide for MSP432 and Getting Started Guide for STM32F4.

This page describes the various commands that a user of the application can use. Each command is a wrapper over a TI's Bluetooth stack API which gets invoked with the parameters selected by the user. This is a subset of the APIs available to the user. TI's Bluetooth stack API documentation (**TI_Bluetooth_Stack_Version-Number\Documentation** or for STM32F4, **TI_Bluetooth_Stack_Version-Number\RTOS_VERSION\Documentation**) describes all of the API's in detail.

Generic Access Profile Commands

Note

Reference the appendix for all Generic Access Profile Commands

Cycling Speed and Cadence Profile Commands

RegisterCSCS

Description

RegisterCSCS is responsible for registering a CSCP Service.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Successfully registered CSCP service

(-4) Function_Error

(-1000) CSCS_ERROR_INVALID_PARAMETER

(-1001) CSCS_ERROR_INSUFFICIENT_RESOURCES

(-1002) CSCS_ERROR_SERVICE_ALREADY_REGISTERED

API Call

CSCS_Initialize_Service(BluetoothStackID, CSCS_EventCallback, NULL, &CSCSInstanceID)

API Prototype

int BTPSAPI CSCS_Initialize_Service(unsigned int BluetoothStackID, CSCS_Event_Callback_t EventCallback, unsigned long CallbackParameter, unsigned int *ServiceID)

Description of API

This function is responsible for opening a CSCS Server.The first parameter is the Bluetooth Stack ID on which to open the Server. The second parameter is the Callback function to call when an event occurs on this Server Port. The third parameter is a user-defined callback parameter that will be passed to the callback function with each event. The final parameter is a pointer to store the GATT Service ID of the registered CSCS service. This can be used to include the service registered by this call. This function returns the positive, non-zero, Instance ID or a negative error code.

UnRegisterCSCS

Description

UnRegisterCSCS is responsible for unregistering a CSCP Service.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Successfully unregistered CSCP service

(-4) Function_Error

(-1000) CSCS_ERROR_INVALID_PARAMETER

(-1003) CSCS_ERROR_INVALID_INSTANCE_ID

API Call

CSCS_Cleanup_Service(BluetoothStackID, CSCSInstanceID)

API Prototype

int BTPSAPI CSCS_Cleanup_Service(unsigned int BluetoothStackID, unsigned int InstanceID)

Description of API

This function is responsible for closing a previously opened CSCS Server. The first parameter is the Bluetooth Stack ID on which to close the Server. The second parameter is the InstanceID that was returned from a

successfull call to CSCS_Initialize_Service(). This function returns a zero if successful or a negative return error code if an error occurs.

SetSupportedFeatures

Description

SetSupportedFeatures is responsible for Setting the supported features for the CSCP Service. This function Can Enable or Disable the support for the next feature: Wheel Revolutions, Crank Revolutions and Multiple Sensor Location. This function will return zero on successful execution and a negative value on errors.

Parameters

There are three parameters for this command:

Wheel Revolution [0 = Disable, 1 = Enable]

Crank Revolution [0 = Disable, 1 = Enable]

Multiple Sensor Location [0 = Disable, 1 = Enable]

Possible Return Values

(0) Success

(-4) Function_Error

(-8) INVALID_STACK_ID_ERROR

(-1004) CSCS_ERROR_INVALID_INSTANCE_ID

(-1008) CSCS_ERROR_WHEEL_NOT_SUPPORTED

(-1009) CSCS_ERROR_CRANK_NOT_SUPPORTED

(-1010) CSCS_ERROR_MULTIPLE_LOCATION_NOT_SUPPORTED

(-1011) CSCS_ERROR_WHEEL_AND_CRANK_NOT_SUPPORTED

(-1012) CSCS_ERROR_WHEEL_AND_MULTIPLE_LOCATION_NOT_SUPPORTED

(-1013) CSCS_ERROR_CRANK_AND_MULTIPLE_LOCATION_NOT_SUPPORTED

(-1014) CSCS_ERROR_WHEEL_AND_CRANK_AND_MULTIPLE_LOCATION_NOT_SUPPORTED

(-1015) CSCS_ERROR_LOCATION_LIST_IS_GREATER_THEN_MAXIMUM_SIZE

API Call

CSCS_Set_Supported_Features(BluetoothStackID, CSCSInstanceID, Features)

API Prototype

int BTPSAPI CSCS_Set_Supported_Features(unsigned int BluetoothStackID, unsigned int InstanceID, Word_t SupportedFeaturesMask)

Description of API

This function is responsible for setting Supported features on the specified CSCS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to CSCS_Initialize_Server(). The final parameter is the Supported Feature bit mask to set the supported Features for the specified CSCS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

Note

The SupportedFeaturesMask is a bit mask that is made up of bit masks of the form CSCS_CSC_FEATURE_BIT_MASK_XXX.

SetSupportedSensorLocationBitMask

Description

SetSupportedSensorLocationBitMaskis responsible for Setting the supported sensor locations for the CSCP Service. This functions is a BitMask from 0x0000 (All Locations are not supported) to 0x7fff(All Locations are supported) This function will return zero on successful execution and a negative value on errors.

Parameters

```
Bit  Category
---  --------
 00  Other
 01  Top of shoe
 02  In shoe
 03  Hip
 04  Front Wheel
 05  Left Crank
 06  Right Crank
 07  Left Pedal
 08  Right Pedal
 09  Front Hub
 10  Rear Dropout
 11  Chainstay
 12  Rear Wheel
 13  Rear Hub
 14  Chest
```

Possible Return Values

(0) Success

(-4) Function_Error

(-8) INVALID_STACK_ID_ERROR

(-1000) CSCS_ERROR_INVALID_PARAMETER

(-1013) CSCS_ERROR_CRANK_AND_MULTIPLE_LOCATION_NOT_SUPPORTED

API Call

CSCS_Set_Sensor_Location_List(BluetoothStackID, CSCSInstanceID, SensorLocatoinBitMask)

API Prototype

int BTPSAPI CSCS_Set_Sensor_Location_List(unsigned int BluetoothStackID, unsigned int InstanceID, Word_t SensorListBitMask)

Description of API

The following function is responsible for setting the supported sensor location list on the Sensor. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to CSCS_Initialize_Server(). The final parameter is the Supported sensor location list bit mask to set on the sensor. This function returns a zero if successful or a negative return error code if an error occurs.

Note

The SensorListBitMask is a bit mask that is made up of bit masks that needs to be smaller then 0x7fff.

QueryFeatures

Description

QueryFeatures is provided to allow a means of reading the supported features on the CSCP Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Success

API Call

CSCS_Query_Supported_Features(BluetoothStackID, CSCSInstanceID, &Features)

API Prototype

int BTPSAPI CSCS_Query_Supported_Features(unsigned int BluetoothStackID, unsigned int InstanceID, Word_t *Features)

Description of API

This function is responsible for querying Supported features on the specified CSCS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to CSCS_Initialize_Server(). The final parameter is a pointer to store the Feature bit mask for the specified CSCS Instance. This function returns the BitMask of the supported Features if successful or a negative return error code if an error occurs.

Note

The Features is a pointer to a bit mask that will be made up of bit masks of the form CSCS_CSC_FEATURE_BIT_MASK_XXX, if this function returns success.

QueryLocation

Description

QueryLocation is provided to allow a means of reading the Location on the CSCP Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Success

(-1000) CSCS_ERROR_INVALID_PARAMETER

(-1004) CSCS_ERROR_INVALID_INSTANCE_ID

API Call

CSCS_Query_Sensor_Location(BluetoothStackID, CSCSInstanceID, &Sensor_Location)

API Prototype

int BTPSAPI CSCS_Query_Sensor_Location(unsigned int BluetoothStackID, unsigned int InstanceID, Byte_t *Sensor_Location)

Description of API

This function is responsible for querying the current Location of the sensor on the specified CSCS Instance. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to CSCS_Initialize_Server(). The final parameter is a pointer to return the current Sensor Location for the specified CSCS Instance. This function returns a zero if successful or a negative return error code if an error occurs.

NotifyMeasurements

Description

NotifyMeasurements is responsible for performing a Measurement notification to a connected remote device. This function notifies measurements for Wheel Cumulative value and Wheel Time Event value or Crank Cumulative value and Crank Time Event value or both. This function will return zero on successful execution and a negative value on errors.

Parameters

When We Support Wheel Revolutions: There are two parameters for this command, They are Wheel Cumulative Value [4Byte], Wheel Event Time Value [2Byte].

When We Support Crank Revolutions: There are two parameters for this command, They are Crank Cumulative Value [2Byte], Crank Event Time Value [2Byte].

When we support both Wheel and Crank Revolutions: All four parameters needs to be entered for this command [4Byte] [2Byte] [2Byte] [2Byte].

Possible Return Values

(0) Sending New CSC Measurements Notification Success

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) CSCS_ERROR_INVALID_PARAMETER

(-1002) CSCS_ERROR_INSUFFICIENT_RESOURCES

(-1004) CSCS_ERROR_INVALID_INSTANCE_ID

(-1005) CSCS_ERROR_MALFORMATTED_DATA

API Call

CSCS_Measurements_Notification(BluetoothStackID, CSCSInstanceID, ConnectionID, &CSCMeasurements)

API Prototype

int BTPSAPI CSCS_Measurements_Notification(unsigned int BluetoothStackID, unsigned int InstanceID, unsigned int ConnectionID, CSCS_Measurements_Data_t *CSCS_Measurement)

Description of API

This function is responsible for sending a Measurement notification to a specified remote device. The first parameter is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to CSCS_Initialize_Server(). The third parameter is the ConnectionID of the remote device to send the notification to. The final parameter is the measurement data to notify. This function returns a zero if successful or a negative return error code if an error occurs.

Note

Mandatory: At least one flag needs to be set in order to send the Notification, otherwise an error will be returend to the caller. When flag WHEEL_REVOLUTION_DATA_PRESENT is set in the flags of the CSCS_Measurements parameter, then Cumulative Wheel Revolutions and Last Wheel Event Time fields are present When flag CRANK_REVOLUTION_DATA_PRESENT is set in the flags of the CSCS_Measurements parameter, then Cumulative Crank Revolutions and Last Crank Event Time fields are present.

QueryNumberAttributes

Description

QueryNumberAttributes is provided to allow a means of reading the amount of attributes on the CSCP Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Success

API Call

CSCS_Query_Number_Attributes()

API Prototype

unsigned int BTPSAPI CSCS_Query_Number_Attributes(void)

Description of API

This function is responsible for querying the number of attributes that are contained in the CSCS Service that is function registered with a call to CSCS_Initialize_Service(). This returns the non-zero number of attributes that are contained in a CSCS Server or zero on failure.

DiscoverCSCS

Description

DiscoverCSCS is responsible for performing a CSCP Service Discovery Operation. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Successfully unregistered CSCP service

(-4) Function_Error

API Call

GATT_Service_Discovery_Start(BluetoothStackID, ConnectionID, (sizeof(UUID)/sizeof(GATT_UUID_t)), UUID, GATT_Event_Callback, sdCSCS)

API Prototype

int BTPSAPI GATT_Service_Discovery_Start(unsigned int BluetoothStackID, unsigned int ConnectionID, unsigned int NumberOfUUID, GATT_UUID_t *UUIDList, GATT_Event_Callback_t ServiceDiscoveryCallback, unsigned long ServiceDiscoveryCallbackParameter)

Description of API

The GATT_Service_Discover_Start is in an application module called GATT that is provided to allow an easy way to perform GATT service discovery. This module can and should be modified for the customers use. This function is called to start a service discovery operation by the GATT module.

GetRemoteFeatures

Description

GetRemoteFeatures is provided to allow a means of performing a read request on a remote device for the supported features on the CSCP Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Successfully unregistered CSCP service

(-4) Function_Error

API Call

GATT_Read_Value_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.CSC_Features, GATT_ClientEventCallback_CSCP, (unsigned long)DeviceInfo->ClientInfo.CSC_Features)

API Prototype

int BTPSAPI GATT_Read_Value_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, GATT_Client_Event_Callback_t ClientEventCallback, unsigned long CallbackParameter)

Description of API

This function is provided to allow a means of performing a read request on a remote device for a specific attribute value. The first parameter is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the attribute handle to read the value from. The final two parameters specify the GATT Client event callback function and callback parameter (respectively) that will be called when a response is received from the remote device. This function will return the positive, non-zero, Transaction ID of the request or a negative error code.

Note

If successful, the return value will contain the Transaction ID that can be used to cancel the request.

ConfigureRemoteCSCS

Description

ConfigureRemoteCSCS is responsible for configure a CSCP Service on a remote device. This Function allow the collector to Enable or Disable Notification and indication from the Server, the function run automatically twice, once for Measurements Notification, and once for SC Control Command Indications (If supported). This function will return zero on successful execution and a negative value on errors.

Parameters

There is one parameter for this command, it is [0 = Disable, 1 = Enable].

Possible Return Values

(0) CCCD Configuration Success

(-1) BTPS_ERROR_INVALID_PARAMETER

(-4) FUNCTION_ERROR

API Call

GATT_Write_Request(BluetoothStackID, ConnectionID, ClientConfigurationHandle, sizeof(Buffer), &Buffer, ClientEventCallback, ClientConfigurationHandle)

API Prototype

int BTPSAPI GATT_Write_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue, GATT_Client_Event_Callback_t ClientEventCallback, unsigned long CallbackParameter)

Description of API

This function is provided to allow a means of performing a write request to a remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write the value of, followed by the length of the value (in bytes), followed by the the actual value data to write. The final two parameters specify the GATT Client event callback function and callback parameter (respectively) that will be called when a response is received from the remote device. This function will return the positive, non-zero, Transaction ID of the request or a negative error code.

Note

This function will not write an attribute value with a length greater than the current MTU - 3. To write a longer attribute value use the GATT_Prepare_Write_Request() function instead. If successful, the return value will contain the Transaction ID that can be used to cancel the request.

GetConfigureRemoteCSCS

Description

GetConfigureRemoteCSCSis responsible for reading the configuration of CSCP Service on a remote device. This function allow the collector read the CCCD of CSC Measurement to check if Notification is enable or disable and the CCCD of SC Control Point to check if the indication is enable or disable, the function check one of the CCCD according to the input. This function will return zero on successful execution and a negative value on errors.

Parameters

There is one parameter for this command, they are [1 = CSC Measurement CCCD, 2 = SC Control Point CCCD].

Possible Return Values

(0) Attempting to read Remote CSC Measurement CCCD

(-4) FUNCTION_ERROR

API Call

GATT_Read_Value_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.CSC_Measurement_Client_Configuration, GATT_ClientEventCallback_CSCP, (unsigned long)DeviceInfo->ClientInfo.CSC_Measurement_Client_Configuration)

API Prototype

int BTPSAPI GATT_Read_Value_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, GATT_Client_Event_Callback_t ClientEventCallback, unsigned long CallbackParameter)

Description of API

This function is provided to allow a means of performing a read request on a remote device for a specific attribute value. The first parameter is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the attribute handle to read the value from. The final two parameters specify the GATT Client event callback function and callback parameter (respectively) that will be called when a response is received from the remote device. This function will return the positive, non-zero, Transaction ID of the request or a negative error code.

Note

If successful, the return value will contain the Transaction ID that can be used to cancel the request.

SetSCControlPoint

Description

SetSCControlPoint is responsible for performing an SC Control Point Command to a connected remote device. The SC Control Point is supported only when Wheel revolutions, Multiple Sensor Location or both are supported. This function allow us to set new wheel cumulative value, set sensor location and retrieve available list location of the remote device. This function will return zero on successful execution and a negative value on errors.

Parameters

There are 3 available op codes for this function When We Support Wheel Revolutions:

Op code [1 = SET CUMULATIVE VALUE], Cumulative Value [Up to 4294967295]

When We Support Multiple Sensor Location we can use the next op codes:

Op Code [2 = UPDATE SENSOR LOCATION], New Location [From Available Locations]

Op Code [3 = REQUEST SUPPORTED SENSOR LOCATION]

Possible Return Values

(0) Sending New SC Control Point command Success

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) CSCS_ERROR_INVALID_PARAMETER

(-1005) CSCS_ERROR_MALFORMATTED_DATA

API Call

CSCS_Format_Control_Point_Command((unsigned int)CSCS_CONTROL_POINT_DATA_SIZE, &Data, &SCControlPoint)

API Prototype

int BTPSAPI CSCS_Format_Control_Point_Command(unsigned int BufferLength, Byte_t *Buffer, CSCS_Control_Point_Data_t *CSCS_Control_Point)

Description of API

The following function is responsible for formatting a Cycling Speed and Cadence SC Control Point Command into a user specified buffer. The first two parameters contain the length of the buffer, and the buffer,to format the command into. The final parameter is the command to format. This function returns a zero if successful or a negative return error code if an error occurs.

Note

The BufferLength and Buffer parameter must point to a buffer of at least CSCS_CONTROL_POINT_DATA_SIZE in size.

API Call

GATT_Write_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.SC_Control_Point, (Word_t)CSCS_CONTROL_POINT_DATA_SIZE, (void *)&Data, GATT_ClientEventCallback_CSCP, DeviceInfo->ClientInfo.SC_Control_Point);

API Prototype

int BTPSAPI GATT_Write_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, Word_t AttributeLength, void *AttributeValue, GATT_Client_Event_Callback_t ClientEventCallback, unsigned long CallbackParameter)

Description of API

This function is provided to allow a means of performing a write request to a remote device for a specified attribute. The first parameter to this function is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the handle of the attribute to write the value of, followed by the length of the value (in bytes), followed by the the actual value data to write. The final two parameters specify the GATT Client event callback function and callback parameter (respectively) that will be called when a response is received from the remote device. This function will return the positive, non-zero, Transaction ID of the request or a negative error code.

Note

This function will not write an attribute value with a length greater than the current MTU - 3. To write a longer attribute value use the GATT_Prepare_Write_Request() function instead. If successful, the return value will contain the Transaction ID that can be used to cancel the request.

SetWheelCircumference

Description

SetWheelCircumference is provided to save the Wheel Circumference parameter at the Client. This function will return zero on successful execution and a negative value on errors.

Parameters

There is one parameter for this command, the value of the wheel circumference [xxx(cm)].

Possible Return Values

(0) Success

(-4) FUNCTION_ERROR

DisplayAvailableLocations

Description

DisplayAvailableLocations is provided to print all the available locations that were retrieved from remote device. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Success

GetRemoteLocation

Description

GetRemoteLocation is provided to allow a means of performing a read request on a remote device for the location on the CSCP Service. This function will return zero on successful execution and a negative value on errors.

Parameters

It is not necessary to include parameters when using this command. A parameter will have no effect on the outcome.

Possible Return Values

(0) Success

(-4) FUNCTION_ERROR

API Call

GATT_Read_Value_Request(BluetoothStackID, ConnectionID, DeviceInfo->ClientInfo.CSC_Sensor_Location, GATT_ClientEventCallback_CSCP, (unsigned long)DeviceInfo->ClientInfo.CSC_Sensor_Location)

API Prototype

int BTPSAPI GATT_Read_Value_Request(unsigned int BluetoothStackID, unsigned int ConnectionID, Word_t AttributeHandle, GATT_Client_Event_Callback_t ClientEventCallback, unsigned long CallbackParameter)

Description of API

This function is provided to allow a means of performing a read request on a remote device for a specific attribute value. The first parameter is the Bluetooth stack ID of the local Bluetooth stack, followed by the connection ID of the connected remote device, followed by the attribute handle to read the value from. The final two parameters specify the GATT Client event callback function and callback parameter (respectively) that will be called when a response is received from the remote device. This function will return the positive, non-zero, Transaction ID of the request or a negative error code.

Note

If successful, the return value will contain the Transaction ID that can be used to cancel the request.

IndicateSCControlPoint

Description

IndicateSCControlPoint is responsible for performing a SC Control Point indication to a connected remote device. This function allows us to check if the indication works, it allows us to simulate an indication response to fake SC Control Point This function will return zero on successful execution and a negative value on errors.

Parameters

There are two parameters for this command Op Code Response [1 = SET CUMULATIVE VALUE, 2 = UPDATE SENSOR LOCATION, 3 = REQUEST SUPPORTED SENSOR LOCATION]

Response Value [1 = SUCCESS, 2 = OP CODE NOT SUPPORTED, 3 = INVALID PARAMETER, 4 = OPERATION FAILED]

Possible Return Values

(0) Success

(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR

(-8) INVALID_STACK_ID_ERROR

(-1000) CSCS_ERROR_INVALID_PARAMETER

(-1002) CSCS_ERROR_INSUFFICIENT_RESOURCES

(-1004) CSCS_ERROR_INVALID_INSTANCE_ID

(-1005) CSCS_ERROR_MALFORMATTED_DATA

(-1006) CSCS_ERROR_INDICATION_IN_PROGRESS

(-1015) CSCS_ERROR_LOCATION_LIST_IS_GREATER_THEN_MAXIMUM_SIZE

API Call

CSCS_SC_Control_Point_Indication(BluetoothStackID, CSCSInstanceID, ConnectionID, &SCControlPoint)

API Prototype

int BTPSAPI CSCS_SC_Control_Point_Indication(unsigned int BluetoothStackID, unsigned int InstanceID, unsigned int ConnectionID, CSCS_Control_Point_Data_t *Op_Code_Response)

Description of API

The following function is responsible for sending a SC Control Point indication to a specified remote device. The first is the Bluetooth Stack ID of the Bluetooth Device. The second parameter is the InstanceID returned from a successful call to CSCS_Initialize_Server(). The third parameter is the ConnectionID of the remote device to send the indication to. The fourth parameter is the Op_Code_Response, the structure who store the data for the indication. This function returns a zero if successful or a negative return error code if an error occurs.

22 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

DATE	REVISION	NOTES
Month Year	*	Initial Release

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265