CC256x TI Bluetooth Stack SPPDemo App

Return to CC256x MSP430 TI's Bluetooth stack Basic Demo APPS (http://processors.wiki.ti.com/index.php/CC256x_MSP430_TI_Bluetooth_Stack#Demos)

Return to CC256x Tiva TI's Bluetooth stack Basic Demo APPS (http://processors.wiki.ti.com/index.php/CC256x_Tiva_TI_Bluetooth_Stack#Demos)

Return to CC256x MSP432 TI's Bluetooth stack Basic Demo APPS (http://www.ti.com/lit/ug/swru453a/swru453a.pdf)

Return to CC256x STM32F4 TI's Bluetooth stack Basic Demo APPS (http://www.ti.com/lit/ug/swru428/swru428.pdf)

Contents

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 (https://store.ti.com/PAN1323EMK-PAN1323-Bluetooth-Evaluation-Module-Kit-P2702.aspx), MSP-EXP430F5438 board (http://www.ti.com/tool/msp-exp430f5438), Tiva DK-TM4C129X (http://www.ti.com/tool/dk-tm4c129x), MSP432 (http://www.ti.com/tool/msp-exp432p401r) or STM32F4 (http://www.st.com/en/evaluation-tools/stm3240g-eval.html). The SPP sample application code is provided to enable a rich out-of-box experience to the user.

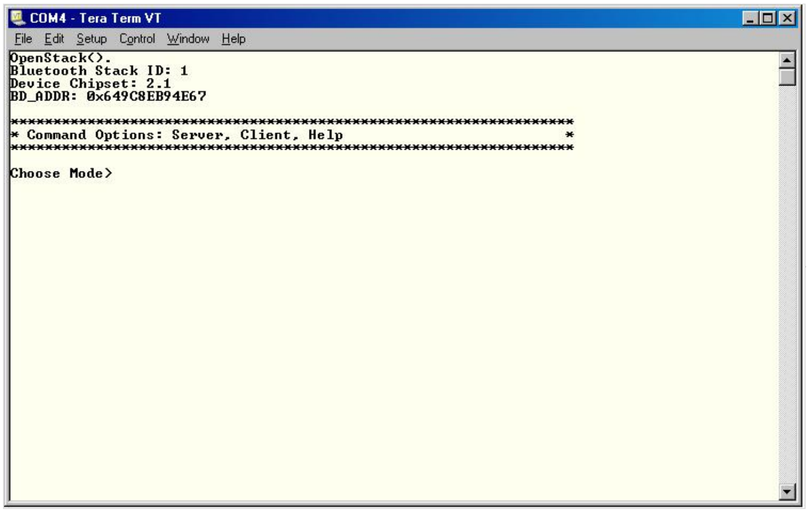
- The application allows the user to use a console to send Bluetooth commands, setup a Bluetooth Device to accept connections, connect to a remote Bluetooth device and communicate over Bluetooth.

For information of the LE version of this profile, refer to the document SPP-LE profile (http://processors.wiki.ti.com/index.php/CC256x_MSP430_TI’s_Bluetooth_Stack_Basic_SPP_SPPLEDemo_APP).

It is recommended that the user visits the kit setup Getting Started Guide for MSP430 (http://processors.wiki.ti.com/index.php/CC256x_MSP430_TI's_Bluetooth_Stack_Basic_Demo_APPS), Getting Started Guide for TIVA (http://processors.wiki.ti.com/index.php/TIVA_TI's_Bluetooth_Stack_Basic_Demo_APPS), Getting Started Guide for MSP432 (http://www.ti.com/lit/ug/swru453a/swru453a.pdf) or Getting Started Guide for STM32F4 (http://www.ti.com/lit/ug/swru428/swru428.pdf) 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.



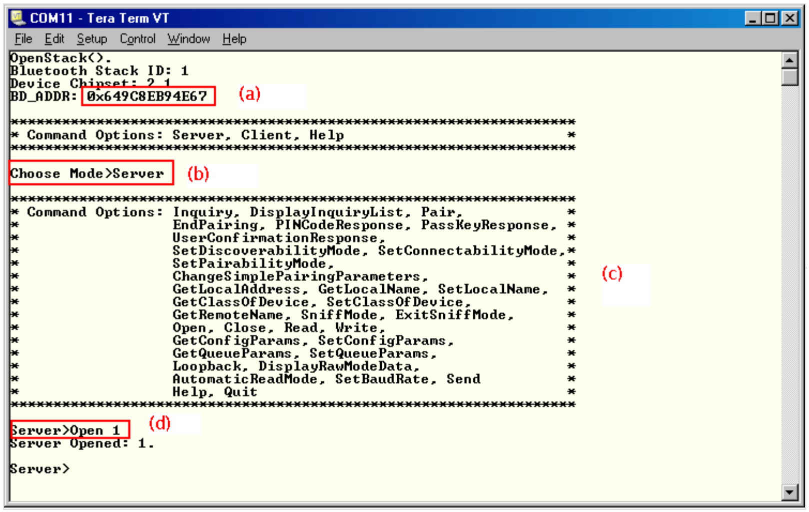
```
OpenStack().
Bluetooth Stack ID: 1
Device Chipset: 2.1
BD_ADDR: 0x649C8EB94E67

******************************************************************
* Command Options: Server, Client, Help                          *
******************************************************************

Choose Mode>
```

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

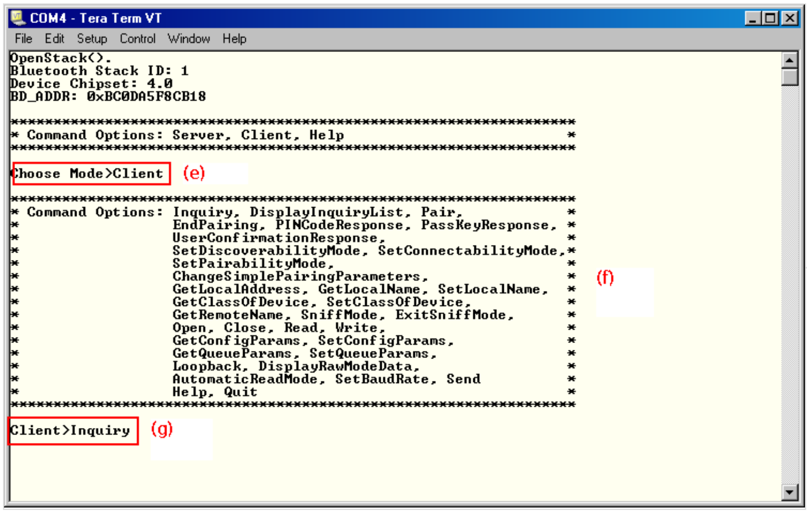
a) We will setup the first board as a server. Perform the steps mentioned earlier in "Running the Bluetooth Code" section to initialize the application. Once initialized, note the Bluetooth address of the server. We will later use this to initiate a connection from the client.
b) On the "Choose mode>" prompt, enter **Server**.
c) 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.
d) 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.



```
OpenStack().
Bluetooth Stack ID: 1
Device Chipset: 2.1
BD_ADDR: 0x649C8EB94E67

******************************************************************
* Command Options: Server, Client, Help                          *
******************************************************************

Choose Mode>Server

******************************************************************
* Command Options: Inquiry, DisplayInquiryList, Pair,            *
*                  EndPairing, PINCodeResponse, PassKeyResponse, *
*                  UserConfirmationResponse,                     *
*                  SetDiscoverabilityMode, SetConnectabilityMode,*
*                  SetPairabilityMode,                           *
*                  ChangeSimplePairingParameters,                *
*                  GetLocalAddress, GetLocalName, SetLocalName,  *
*                  GetClassOfDevice, SetClassOfDevice,           *
*                  GetRemoteName, SniffMode, ExitSniffMode,      *
*                  Open, Close, Read, Write,                     *
*                  GetConfigParams, SetConfigParams,             *
*                  GetQueueParams, SetQueueParams,               *
*                  Loopback, DisplayRawModeData,                 *
*                  AutomaticReadMode, SetBaudRate, Send          *
*                  Help, Quit                                    *
******************************************************************

Server>Open 1
Server Opened: 1.

Server>
```

Client setup and device discovery on the demo application

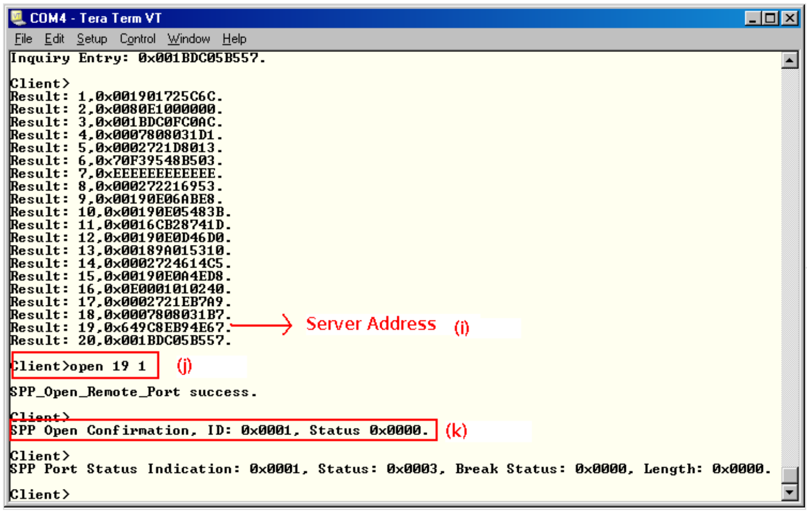
e) We will setup the second board as a client. Perform the steps mentioned earlier in "Running the Bluetooth Code" section to initialize the application. On the "Choose mode>" prompt, enter **Client**.
f) You will see a list of all possible commands at this time for a Client. You can see this list at any time by entering **Help** at the Client> prompt.
g) 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.
h) You can access this list any time by choosing **DisplayInquiryList** at the Client prompt.



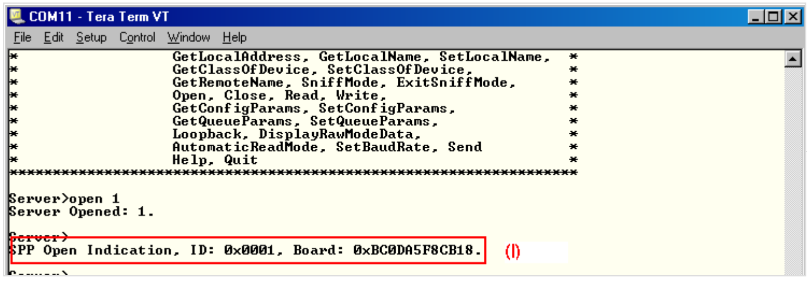
```
OpenStack().
Bluetooth Stack ID: 1
Device Chipset: 4.0
BD_ADDR: 0xBC0DA5F8CB18

******************************************************************
* Command Options: Server, Client, Help                          *
******************************************************************

Choose Mode>Client

******************************************************************
* Command Options: Inquiry, DisplayInquiryList, Pair,            *
*                  EndPairing, PINCodeResponse, PassKeyResponse, *
*                  UserConfirmationResponse,                     *
*                  SetDiscoverabilityMode, SetConnectabilityMode,*
*                  SetPairabilityMode,                           *
*                  ChangeSimplePairingParameters,                *
*                  GetLocalAddress, GetLocalName, SetLocalName,  *
*                  GetClassOfDevice, SetClassOfDevice,           *
*                  GetRemoteName, SniffMode, ExitSniffMode,      *
*                  Open, Close, Read, Write,                     *
*                  GetConfigParams, SetConfigParams,             *
*                  GetQueueParams, SetQueueParams,               *
*                  Loopback, DisplayRawModeData,                 *
*                  AutomaticReadMode, SetBaudRate, Send          *
*                  Help, Quit                                    *
******************************************************************

Client>Inquiry
```

Initiating connection from the client

i)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.]
j) Issue a **Open <index number> <server port number>** command at the command prompt.
k) Wait for SPP Open confirmation.
l) When a client successfully connects to a server, the server will see the open indication.



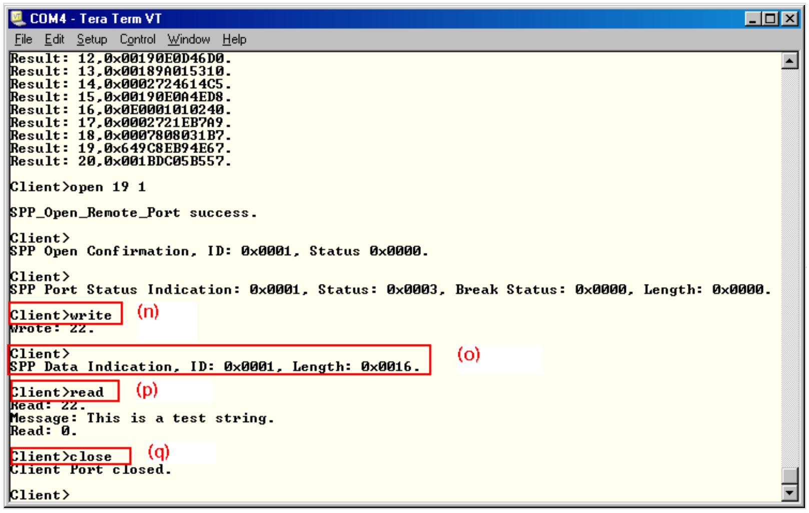
```
Inquiry Entry: 0x001BDC05B557.

Client>
Result: 1,0x001901725C6C.
Result: 2,0x0080E1000000.
Result: 3,0x001BDC0FC0AC.
Result: 4,0x0007808031D1.
Result: 5,0x0002721D8013.
Result: 6,0x70F39548B503.
Result: 7,0xEEEEEEEEEEEE.
Result: 8,0x000272216953.
Result: 9,0x00190E06ABE8.
Result: 10,0x00190E05483B.
Result: 11,0x0016CB28741D.
Result: 12,0x00190E0D46D0.
Result: 13,0x00189A015310.
Result: 14,0x0002724614C5.
Result: 15,0x00190E0A4ED8.
Result: 16,0x0E0001010240.
Result: 17,0x0002721EB7A9.
Result: 18,0x0007808031B7.
Result: 19,0x649C8EB94E67.
Result: 20,0x001BDC05B557.

Client>open 19 1

SPP_Open_Remote_Port success.

Client>
SPP Open Confirmation, ID: 0x0001, Status 0x0000.

Client>
SPP Port Status Indication: 0x0001, Status: 0x0003, Break Status: 0x0000, Length: 0x0000.

Client>
```



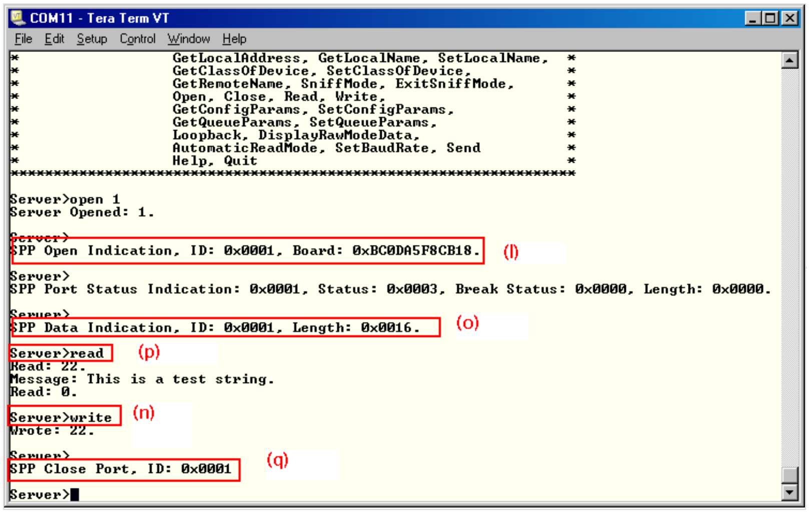
```
*                   GetLocalAddress, GetLocalName, SetLocalName,   *
*                   GetClassOfDevice, SetClassOfDevice,             *
*                   GetRemoteName, SniffMode, ExitSniffMode,        *
*                   Open, Close, Read, Write,                       *
*                   GetConfigParams, SetConfigParams,               *
*                   GetQueueParams, SetQueueParams,                 *
*                   Loopback, DisplayRawModeData,                   *
*                   AutomaticReadMode, SetBaudRate, Send            *
*                   Help, Quit                                      *
*******************************************************************

Server>open 1
Server Opened: 1.

Server>
SPP Open Indication, ID: 0x0001, Board: 0xBC0DA5F8CB18.
```

Data Transfer between Client and Server

m) Now we have a SPP connection established and both devices are ready to transmit data to each other.
n) 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.
o) The remote side will receive a data indication
p) The user can read the data by issuing a **Read** command.
q) 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.



```
Result: 12,0x00190E0D46D0.
Result: 13,0x00189A015310.
Result: 14,0x0002724614C5.
Result: 15,0x00190E0A4ED8.
Result: 16,0x0E0001010240.
Result: 17,0x0002721EB7A9.
Result: 18,0x0007808031B7.
Result: 19,0x649C8EB94E67.
Result: 20,0x001BDC05B557.

Client>open 19 1

SPP_Open_Remote_Port success.

Client>
SPP Open Confirmation, ID: 0x0001, Status 0x0000.

Client>
SPP Port Status Indication: 0x0001, Status: 0x0003, Break Status: 0x0000, Length: 0x0000.

Client>write
Wrote: 22.

Client>
SPP Data Indication, ID: 0x0001, Length: 0x0016.

Client>read
Read: 22.
Message: This is a test string.
Read: 0.

Client>close
Client Port closed.

Client>
```



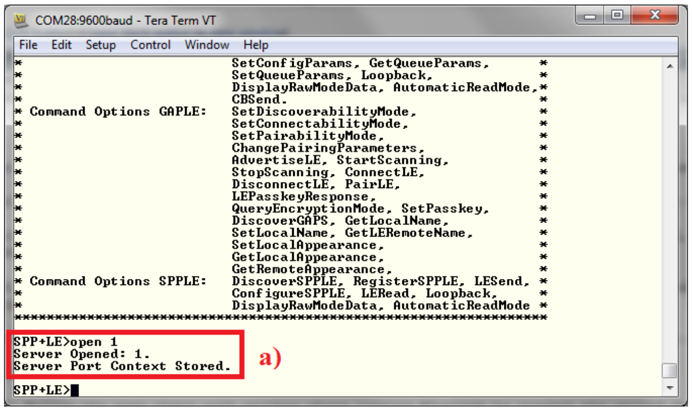
```
*                    GetLocalAddress, GetLocalName, SetLocalName,   *
*                    GetClassOfDevice, SetClassOfDevice,            *
*                    GetRemoteName, SniffMode, ExitSniffMode,       *
*                    Open, Close, Read, Write,                      *
*                    GetConfigParams, SetConfigParams,              *
*                    GetQueueParams, SetQueueParams,                *
*                    Loopback, DisplayRawModeData,                  *
*                    AutomaticReadMode, SetBaudRate, Send           *
*                    Help, Quit                                     *
*********************************************************************

Server>open 1
Server Opened: 1.

Server>
SPP Open Indication, ID: 0x0001, Board: 0xBC0DA5F8CB18.

Server>
SPP Port Status Indication: 0x0001, Status: 0x0003, Break Status: 0x0000, Length: 0x0000.

Server>
SPP Data Indication, ID: 0x0001, Length: 0x0016.

Server>read
Read: 22.
Message: This is a test string.
Read: 0.

Server>write
Wrote: 22.

Server>
SPP Close Port, ID: 0x0001

Server>
```

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 (https:///play.google.com/store/apps/details?id=es.pymasde.blueterm&hl=en) .

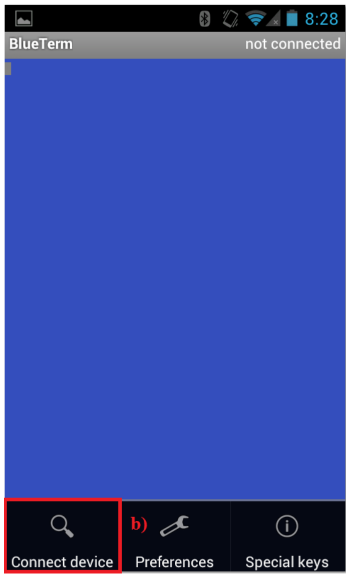
Note: The app requires Android V2.1 and above

a) Open a server port on the MSP430.

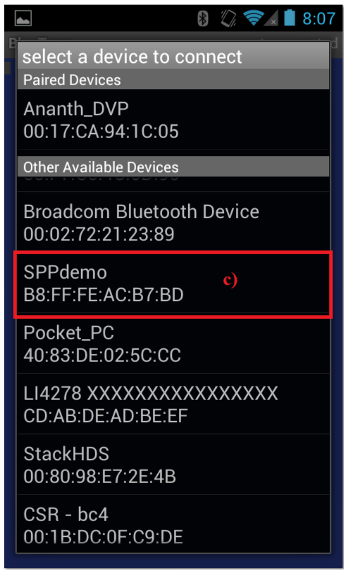

COM28:9600baud - Tera Term VT
File Edit Setup Control Window Help
a)

```
*                          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>open 1
Server Opened: 1.
Server Port Context Stored.

SPP+LE>
```

b) To connect to the device, open blueterm, press the menu button and hit the **connect** icon.

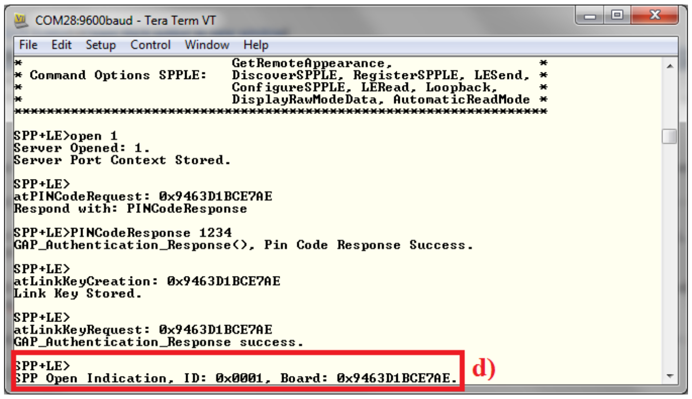


c) The app should show a list of paired devices. If the device is not already paired, select scan. It should search for available bluetooth devices.



```
8:07
select a device to connect
Paired Devices
Ananth_DVP
00:17:CA:94:1C:05
Other Available Devices
Broadcom Bluetooth Device
00:02:72:21:23:89
SPPdemo
B8:FF:FE:AC:B7:BD
Pocket_PC
40:83:DE:02:5C:CC
LI4278 XXXXXXXXXXXXXXXX
CD:AB:DE:AD:BE:EF
StackHDS
00:80:98:E7:2E:4B
CSR - bc4
00:1B:DC:0F:C9:DE
```

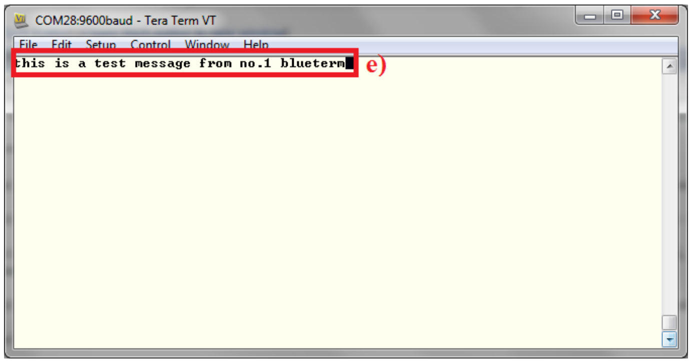
d) Select and Pair it with the device running the SPPdemo. If it needs a pincode enter a pincode on the app/phone. The MSP430 will prompt for a **PINCodeResponse**. Type **PINCodeResponse <the pin code you used for the phone>** in the prompt. It will pair and you should see an open indication on the MSP430.



COM28:9600baud - Tera Term VT
File Edit Setup Control Window Help
d)

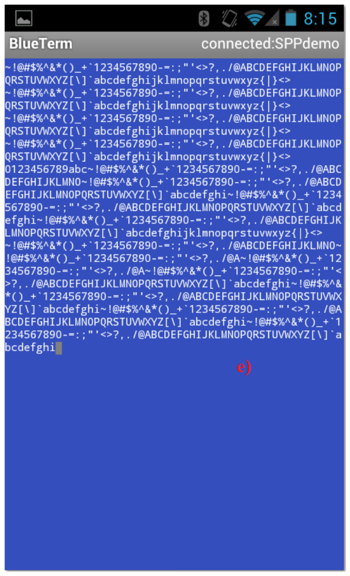
```
*                           GetRemoteAppearance,                    *
* Command Options SPPLE:    DiscoverSPPLE, RegisterSPPLE, LESend,   *
*                           ConfigureSPPLE, LERead, Loopback,       *
*                           DisplayRawModeData, AutomaticReadMode   *
*********************************************************************

SPP+LE>open 1
Server Opened: 1.
Server Port Context Stored.

SPP+LE>
atPINCodeRequest: 0x9463D1BCE7AE
Respond with: PINCodeResponse

SPP+LE>PINCodeResponse 1234
GAP_Authentication_Response(), Pin Code Response Success.

SPP+LE>
atLinkKeyCreation: 0x9463D1BCE7AE
Link Key Stored.

SPP+LE>
atLinkKeyRequest: 0x9463D1BCE7AE
GAP_Authentication_Response success.

SPP+LE>
SPP Open Indication, ID: 0x0001, Board: 0x9463D1BCE7AE.
```

e) The two devices are now connected. Data can be sent/received from the two devices as shown here.



COM28:9600baud - Tera Term VT
File Edit Setup Control Window Help
e)

```
this is a test message from no.1 blueterm
```



```
~!@#$%^&*()_+`1234567890-=:;"'<>?,./@ABCDEFGHIJKLMNOP
QRSTUVWXYZ[\]`abcdefghijklmnopqrstuvwxyz{|}<>
~!@#$%^&*()_+`1234567890-=:;"'<>?,./@ABCDEFGHIJKLMNOP
QRSTUVWXYZ[\]`abcdefghijklmnopqrstuvwxyz{|}<>
~!@#$%^&*()_+`1234567890-=:;"'<>?,./@ABCDEFGHIJKLMNOP
QRSTUVWXYZ[\]`abcdefghijklmnopqrstuvwxyz{|}<>
~!@#$%^&*()_+`1234567890-=:;"'<>?,./@ABCDEFGHIJKLMNOP
QRSTUVWXYZ[\]`abcdefghijklmnopqrstuvwxyz{|}<>
0123456789abc~!@#$%^&*()_+`1234567890-=:;"'<>?,./@ABC
DEFGHIJKLMNO~!@#$%^&*()_+`1234567890-=:;"'<>?,./@ABCD
EFGHIJKLMNOPQRSTUVWXYZ[\]`abcdefghi~!@#$%^&*()_+`1234
567890-=:;"'<>?,./@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]`abcd
efghi~!@#$%^&*()_+`1234567890-=:;"'<>?,./@ABCDEFGHIJK
LMNOPQRSTUVWXYZ[\]`abcdefghijklmnopqrstuvwxyz{|}<>
~!@#$%^&*()_+`1234567890-=:;"'<>?,./@ABCDEFGHIJKLMNO~
!@#$%^&*()_+`1234567890-=:;"'<>?,./@A~!@#$%^&*()_+`12
34567890-=:;"'<>?,./@A~!@#$%^&*()_+`1234567890-=:;"'<
>?,./@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]`abcdefghi~!@#$%^&
*()_+`1234567890-=:;"'<>?,./@ABCDEFGHIJKLMNOPQRSTUVWX
YZ[\]`abcdefghi~!@#$%^&*()_+`1234567890-=:;"'<>?,./@A
BCDEFGHIJKLMNOPQRSTUVWXYZ[\]`abcdefghi~!@#$%^&*()_+`1
234567890-=:;"'<>?,./@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]`a
bcdefghi
```

e)

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.

Note: 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.

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.

The basic bluetooth application included with MSP-EXP430F5438 (http://www.ti.com/tool/msp-exp430f5438), Tiva DK-TM4C129X (http://www.ti.com/tool/dk-tm4c129x), MSP432 (http://www.ti.com/tool/msp-exp432p401r) and STM32F4 (http://www.st.com/en/evaluation-tools/stm3240g-eval.html) is a Serial Port Profile Application.

An overview of the application and other applications can be read at the Getting Started Guide (http://processors.wiki.ti.com/index.php/CC256x_MSP430_TI's_Bluetooth_Stack_Basic_Demo_APPS)for MSP430, Getting Started Guide (http://processors.wiki.ti.com/index.php/TIVA_TI's_Bluetooth_Stack_Basic_Demo_APPS) for TIVA M4, Getting Started Guide (http://www.ti.com/lit/ug/swru453a/swru453a.pdf) for MSP432 and Getting Started Guide (http://www.ti.com/lit/ug/swru428/swru428.pdf) 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

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.

Inquiry

Description
The Inquiry command is responsible for performing a General Inquiry for discovering Bluetooth Devices. The command requires that a valid Bluetooth Stack ID exists before running. This command returns zero on a successful call or a negative value if an error occurred during execution. The inquiry will last 10 seconds unless 20 devices (MAX_INQUIRY_RESULTS) are found before that time limit.

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

Possible Return Values
(0) Successful Inquiry Procedure
(-1) BTPS_ERROR_INVALID_PARAMETER
(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID
(-57) BTPS_ERROR_DEVICE_HCI_ERROR
(-58) BTPS_ERROR_INVALID_MODE

API Call
GAP_Perform_Inquiry(BluetoothStackID, itGeneralInquiry, 0, 0, 10, MAX_INQUIRY_RESULTS, GAP_Event_Callback, (unsigned long) NULL);

API Prototype
int BTPSAPI GAP_Perform_Inquiry(unsigned int BluetoothStackID, GAP_Inquiry_Type_t GAP_Inquiry_Type, unsigned int MinimumPeriodLength, unsigned int MaximumPeriodLength, unsigned int InquiryLength, unsigned int MaximumResponses, GAP_Event_Callback_t GAP_Event_Callback, unsigned long CallbackParameter);

Description of API
This function is provided to allow a mechanism for starting an Inquiry Scan Procedure. The first parameter to this function is the Bluetooth Protocol Stack of the Bluetooth Device that is to perform the Inquiry. The second parameter is the type of Inquiry to perform. The third and fourth parameters are the Minimum and Maximum Period Lengths which are specified in seconds (only valid in case a Periodic Inquiry is to be performed). The fifth parameter is the Length of Time to perform the Inquiry, specified in seconds. The sixth parameter is the Number of Responses to wait for. The final two parameters represent the Callback Function (and parameter) that is to be called when the specified Inquiry has completed. This function returns zero is successful, or a negative return error code if an Inquiry was unable to be performed. Only ONE Inquiry can be performed at any given time. Calling this function with an outstanding Inquiry is in progress will fail. The caller can call the GAP_Cancel_Inquiry() function to cancel a currently executing Inquiry procedure. The Minimum and Maximum Inquiry Parameters are optional and, if specified, represent the Minimum and Maximum Periodic Inquiry Periods. The called should set BOTH of these values to zero if a simple Inquiry Procedure is to be used (Non-Periodic). If these two parameters are specified, then these two parameters must satisfy the following formula:
MaximumPeriodLength > MinimumPeriodLength > InquiryLength

Pair

Description
The Pair command is responsible for initiating bonding with a remote Bluetooth Device. The function returns zero on a successful execution and a negative value on all errors. A Bluetooth Stack ID must exist before attempting to pair and the device must not already be connected to any device (including the one it tries to pair with). It is also important to note

that the use of the Inquiry command before calling Pair is necessary to connect to a remote device. Both General and Dedicated bonding are supported.

Parameters
The Pair command requires one or two parameters with specific values in order to work successfully. The first parameter is the Inquiry Index of the remote Bluetooth Device. This parameter is always necessary. This 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. The second parameter is the bonding type used for the pairing procedure. It is an optional parameter which is only required if General Bonding is desired for the connection. This must be specified as either 0 (for Dedicated Bonding) or 1 (for General Bonding). If only one parameter is given, the Bonding Type will be Dedicated Bonding.

Command Call Examples
"Pair 5 0" Attempts to pair with the remote device at the fifth Inquiry Index using Dedicated Bonding.
"Pair 5" Is the exact same as the above example. If no parameters, the Bonding Type will be Dedicated.
"Pair 8 1" Attempts to pair with the remote device at the eighth Inquiry Index using General Bonding.

Possible Return Values
(0) Successful Pairing
(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID
(-1) BTPS_ERROR_INVALID_PARAMETER
(-59) BTPS_ERROR_ADDING_CALLBACK_INFORMATION
(-8) BTPS_ERROR_DEVICE_HCI_ERROR

API Call
GAP_Initiate_Bonding(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam – 1)], BondingType, GAP_Event_Callback, (unsigned long)0);

API Prototype
int BTPSAPI GAP_Initiate_Bonding(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, GAP_Bonding_Type_t GAP_Bonding_Type, GAP_Event_Callback_t GAP_Event_Callback, unsigned long CallbackParameter);

Description of API
This function is provided to allow a means to Initiate a Bonding Procedure. This function can perform both General and Dedicated Bonding based upon the type of Bonding requested. This function accepts as input, the Bluetooth Protocol Stack ID of the Local Bluetooth device that is perform the Bonding, the Remote Bluetooth address of the Device to Bond with, the type of bonding to perform, and the GAP Event Callback Information that will be used to handle Authentication Events that will follow if this function is successful. If this function is successful, then all further information will be returned through the Registered GAP Event Callback. It should be noted that if this function returns success that it does NOT mean that the Remote Device has successfully Bonded with the Local Device, ONLY that the Remote Device Bonding Process has been started. This function will only succeed if a Physical Connection to the specified Remote Bluetooth device does NOT already exist. This function will connect to the Bluetooth device and begin the Bonding Process. If General Bonding is specified, then the Link is maintained, and will NOT be terminated until the GAP_End_Bonding function has been called. This will allow any higher level initialization that is needed on the same physical link. If Dedicated Bonding is performed, then the Link is terminated automatically when the Authentication Process has completed.Due to the asynchronous nature of this process, the GAP Event Callback that is specified will inform the caller of any Events and/or Data that is part of the Authentication Process. The GAP_Cancel_Bonding function can be called at any time to end the Bonding Process and terminate the link (regardless of which Bonding method is being performed).When using General Bonding, if an L2CAP Connection is established over the Bluetooth Link that was initiated with this function, the Bluetooth Protocol Stack MAY or MAY NOT terminate the Physical Link when (and if) an L2CAP Disconnect Request (or Response) is issued. If this occurs, then calling the GAP_End_Bonding function will have no effect (the GAP_End_Bonding function will return an error code in this case).

EndPairing

Description
The EndPairing command is responsible for ending a previously initiated bonding session with a remote device. The function returns zero on a successful execution and a negative value on all errors. A Bluetooth Stack ID must exist before attempting to end pairing and the device must already be connected to a remote device. It is also important to note that the use of the Pair and Inquiry commands before calling EndPairing are necessary to disconnect from a remote device.

Parameters
The EndPairing 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. It should be the same value as the first parameter used in the Pair command, unless a new Inquiry has been called after pairing. If this is the case, find the Bluetooth Address of the device used in the Pair command.

Command Call Examples
"EndPairing 5" Attempts to end pairing with the remote device at the fifth Inquiry Index.
"EndPairing 8" Attempts to end pairing with the remote device at the eighth Inquiry Index.

Possible Return Values
(0) Successful End Pairing
(-2)BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID
(-1)BTPS_ERROR_INVALID_PARAMETER
(-58)BTPS_ERROR_INVALID_MODE
(-4) FUNCTION_ERROR
(-6) INVALID_PARAMETERS_ERROR
(-8) INVALID_STACK_ID_ERROR

API Call
GAP_End_Bonding(BluetoothStackID, InquiryResultList[(TempParam->Params[0].intParam – 1)]);

API Prototype
int BTPSAPI GAP_Initiate_Bonding(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, GAP_Bonding_Type_t GAP_Bonding_Type, GAP_Event_Callback_t GAP_Event_Callback, unsigned long CallbackParameter);

Description of API
This function is provided to allow a means to terminate a connection that was established via a call to the GAP_Initiate_Bonding function (that specified general bonding as the bonding type to perform). This function has NO effect if the bonding procedure was initiated using dedicated bonding (or the device is already disconnected). This function accepts the Bluetooth device address of the remote Bluetooth device that was specified to be bonded with (general bonding). This function terminates the ACL connection that was established and it guarantees that NO GAP Event Callbacks will be issued to the GAP Event Callback that was specified in the original GAP_Initiate_Bonding function call (if this function returns success).

PINCodeResponse

Description
The PINCodeResponse command is responsible for issuing a GAP Authentication Response with a PIN Code value specified via the input parameter. 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 device must also be in the middle of an on-going Pairing operation that was started by the local device or a remote device.

Parameters
The PINCodeResponse command requires one parameter which is the PIN Code used for authenticating the connection. This is a string value which can be up to 16 digits long. The initiator of the Pairing will see a message displayed during the Pairing Procedure to call this command. A responder will receive a message to call this command after the initiator has put in the PIN Code.

Command Call Examples
"PINCodeResponse 1234" Attempts to set the PIN Code to "1234."
"PINCodeResponse 5921302312564542 Attempts to set the PIN Code to "5921302312564542." This value represents the longest PIN Code value of 16 digits.

Possible Return Values
(0) Successful PIN Code 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
(-57) BTPS_ERROR_DEVICE_HCI_ERROR

API Call
GAP_Authentication_Response(BluetoothStackID, CurrentRemoteBD_ADDR, &GAP_Authentication_Information);

API Prototype
*int BTPSAPI GAP_Authentication_Response(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, GAP_Authentication_Information_t *GAP_Authentication_Information);*

Description of API
This function is provided to allow a mechanism for the local device to respond to GAP 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).

PassKeyResponse

Description
The PassKeyResponse command is responsible for issuing a GAP Authentication Response with a Pass Key value via the input parameter. 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 device must also be in the middle of an on-going Pairing operation that was started by the local device or a remote device.

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
(-57) BTPS_ERROR_DEVICE_HCI_ERROR

API Call
GAP_Authentication_Response(BluetoothStackID, CurrentRemoteBD_ADDR, &GAP_Authentication_Information);

API Prototype
*int BTPSAPI GAP_Authentication_Response(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, GAP_Authentication_Information_t *GAP_Authentication_Information);*

Description of API
This function is provided to allow a mechanism for the local device to respond to GAP 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).

UserConfirmationResponse

Description
The UserConfirmationResponse command is responsible for issuing a GAP Authentication Response with a User Confirmation value via the input parameter. 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 device must also be in the middle of an on-going Pairing operation that was started by the local device or a remote device.

Parameters
The UserConfirmationResponse command requires one parameter which is the User Confirmation value used for authenticating the connection. This is an integer value that must be either 1, to confirm the connection, or 0 to NOT confirm the Authentication and stop the Pairing Procedure.
Command Call Examples
"UserConfirmationResponse 0" Attempts to decline the connection made with a remote Bluetooth Device and cancels the Authentication Procedure.
"UserConfirmationResponse 1" Attempts to accept the connection made with a remote Bluetooth Device and confirm the Authentication Procedure.
Possible Return Values
(0) Successful User Confirmation 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
(-57) BTPS_ERROR_DEVICE_HCI_ERROR

API Call
GAP_Authentication_Response(BluetoothStackID, CurrentRemoteBD_ADDR, &GAP_Authentication_Information);

API Prototype
*int BTPSAPI GAP_Authentication_Response(unsigned int BluetoothStackID, BD_ADDR_t BD_ADDR, GAP_Authentication_Information_t *GAP_Authentication_Information);*

Description of API
This function is provided to allow a mechanism for the local device to respond to GAP 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).

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. A Bluetooth Stack ID must exist before attempting to call this function. If setting the device as Limited Discoverable, the device will be discoverable for 60 seconds; a General Discoverable device will always be discoverable.

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
(-4) FUNCTION_ERROR

(-6) INVALID_PARAMETERS_ERROR
(-8) INVALID_STACK_ID_ERROR
(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID
(-5) BTPS_ERROR_GAP_NOT_INITIALIZED
(-58) BTPS_ERROR_INVALID_MODE
(-57) BTPS_ERROR_DEVICE_HCI_ERROR
(-64) BTPS_ERROR_INTERNAL_ERROR
(-1) BTPS_ERROR_INVALID_PARAMETER

API Call
GAP_Set_Discoverability_Mode(BluetoothStackID, DiscoverabilityMode, (DiscoverabilityMode == dmLimitedDiscoverableMode)?60:0);

API Prototype
int BTPSAPI GAP_Set_Discoverability_Mode(unsigned int BluetoothStackID, GAP_Discoverability_Mode_t GAP_Discoverability_Mode, unsigned int Max_Discoverable_Time);

Description of API
This function is provided to set the discoverability mode of the local Bluetooth device specified by the Bluetooth Protocol Stack that is specified by the Bluetooth protocol stack ID. The second parameter specifies the discoverability mode to place the local Bluetooth device into, and the third parameter species the length of time (in seconds) that the local Bluetooth device is to be placed into the specified discoverable mode (if mode is not specified as non-discoverable). At the end of this time (provided the time is not infinite), the local Bluetooth device will return to non-discoverable mode.

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. A Bluetooth Stack ID must exist before attempting to call this function.

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-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
(-4) FUNCTION_ERROR
(-6) INVALID_PARAMETERS_ERROR
(-8) INVALID_STACK_ID_ERROR
(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID
(-5) BTPS_ERROR_GAP_NOT_INITIALIZED
(-58) BTPS_ERROR_INVALID_MODE
(-57) BTPS_ERROR_DEVICE_HCI_ERROR

API Call
GAP_Set_Connectability_Mode(BluetoothStackID, ConnectableMode);

API Prototype
int BTPSAPI GAP_Set_Connectability_Mode(unsigned int BluetoothStackID, GAP_Connectability_Mode_t GAP_Connectability_Mode);

Description of API
This function is provided to set the connectability mode of the local Bluetooth device specified by the Bluetooth protocol stack that is specified by the Bluetooth protocol stack ID. The second parameter specifies the connectability mode to place the local Bluetooth device into.

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. A Bluetooth Stack ID must exist before attempting to call this function.

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 Secure Simple Pairing).

Command Call Examples
"SetPairabilityMode 0" Attempts to set the Pairability Mode of the Local Device to Non-Pairable.
"SetPairabilityMode 1" Attempts to set the Pairability Mode of the Local Device to Pairable.

- "SetPairabilityMode 2" Attempts to set the Pairability Mode of the Local Device to Secure Simple Pairing.

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
(-5) BTPS_ERROR_GAP_NOT_INITIALIZED
(-58) BTPS_ERROR_INVALID_MODE

API Call
GAP_Set_Pairability_Mode(BluetoothStackID, PairabilityMode);

API Prototype
int BTPSAPI GAP_Set_Pairability_Mode(unsigned int BluetoothStackID, GAP_Pairability_Mode_t GAP_Pairability_Mode);

Description of API
This function is provided to set the pairability mode of the local Bluetooth device. The second parameter specifies the pairability mode to place the local Bluetooth device into. If secure simple pairing (SSP) pairing mode is specified, then SSP *MUST* be used for all pairing operations. The device can be placed into non pairable mode after this, however, if pairing is re-enabled, it *MUST* be set to pairable with SSP enabled.

GetLocalAddress

Description
The GetLocalAddress command is responsible for querying the Bluetooth Device Address of the local Bluetooth Device. 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.

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 Query Local Address
(-1) BTPS_ERROR_INVALID_PARAMETER
(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID
(-8) INVALID_STACK_ID_ERROR
(-4) FUNCTION_ERROR

API Call
GAP_Query_Local_BD_ADDR(BluetoothStackID, &BD_ADDR);

API Prototype
*int BTPSAPI GAP_Query_Local_BD_ADDR(unsigned int BluetoothStackID, BD_ADDR_t *BD_ADDR);*

Description of API
This function is responsible for querying (and reporting) the device address of the local Bluetooth device. The second parameter is a pointer to a buffer that is to receive the device address of the local Bluetooth device. If this function is successful, the buffer that the BD_ADDR parameter points to will be filled with the device address read from the local Bluetooth device. If this function returns a negative value, then the device address of the local Bluetooth device 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 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.

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.

GetLocalName

Description
This function is responsible for querying the name of the local Bluetooth Device. 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.

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).

SetClassOfDevice

Description
The SetClassOfDevice command is responsible for setting the Class of Device of the local Bluetooth Device to a Class of Device value. 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.

Parameters
The only parameter needed is the new Class of Device value. It is preferred to start the value with “0x” and use a six digit value after that. Without doing this, the Class of Device written will be assumed decimal and will be converted to hexadecimal format and change the values given.

Command Call Examples
"SetClassOfDevice 0x123456" Attempts to set the Class of Device for the local Bluetooth Device to "0x123456."
"SetClassOfDevice 123456" Attempts to set the Class of Device for the local Bluetooth Device to "0x01E240" which is equivalent to the decimal value of 123456.

Possible Return Values
(0) Successfully Set Local Class of Device
(-57) BTPS_ERROR_DEVICE_HCI_ERROR
(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID
(-8) INVALID_STACK_ID_ERROR
(-4) FUNCTION_ERROR
(-5) BTPS_ERROR_GAP_NOT_INITIALIZED

API Call
GAP_Set_Class_of_Device(BluetoothStackID, Class_of_Device);

API Prototype
int BTPSAPI GAP_Set_Class_Of_Device(unsigned int BluetoothStackID, Class_of_Device_t Class_of_Device);

Description of API
This function is provided to allow the changing of the class of device of the local Bluetooth device. The Class_of_Device parameter represents the class of device value that is to be written to the local Bluetooth device. This function will return zero if the class of device was successfully changed, or a negative return error code if there was an error condition.

GetClassOfDevice

Description
The GetClassOfDevice command is responsible for querying the Bluetooth Class of Device of the local Bluetooth Device. 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.

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 Class of Device
(-57) BTPS_ERROR_DEVICE_HCI_ERROR
(-2) BTPS_ERROR_INVALID_BLUETOOTH_STACK_ID
(-8) INVALID_STACK_ID_ERROR
(-4) FUNCTION_ERROR
(-1) BTPS_ERROR_INVALID_PARAMETER

API Call
GAP_Query_Class_Of_Device(BluetoothStackID, &Class_of_Device);

API Prototype
*int BTPSAPI GAP_Query_Class_Of_Device(unsigned int BluetoothStackID, Class_of_Device_t *Class_of_Device);*

Description of API
This function is responsible for querying (and reporting) the class of device of the local Bluetooth device. The second parameter is a pointer to a class of device buffer that is to receive the Bluetooth class of device of the local device. If this function is successful, this function returns zero, and the buffer that Class_Of_Device points to will be filled with the Class of Device read from the local Bluetooth device. If there is an error, this function returns a negative value, and the class of device of the local Bluetooth device is NOT copied into the specified input buffer.

GetRemoteName

Description
The GetRemoteName command is responsible for querying the Bluetooth Device Name of a Remote Device. 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. 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).

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

DisplayInquiryList

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

MemoryUsage (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

{{ 1. switchcategory:MultiCore= • For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum • For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum Please post only comments related to the article **CC256x TI Bluetooth Stack SPPDemo App** here.	Keystone= • For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum • For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum Please post only comments related to the article **CC256x TI Bluetooth Stack SPPDemo App** here.	C2000=*For technical support on the C2000 please post your questions on The C2000 Forum. Please post only comments about the article* ***CC256x TI Bluetooth Stack SPPDemo App*** *here.*	DaVinci=*For technical support on DaVincoplease post your questions on The DaVinci Forum. Please post only comments about the article* ***CC256x TI Bluetooth Stack SPPDemo App*** *here.*	MSP430=*For technical support on MSP430 please post your questions on The MSP430 Forum. Please post only comments about the article* ***CC256x TI Bluetooth Stack SPPDemo App*** *here.*	OMAP35x=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article* ***CC256x TI Bluetooth Stack SPPDemo App*** *here.*	OMAPL1=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article* ***CC256x TI Bluetooth Stack SPPDemo App*** *here.*	MAVRK=*For technical support on MAVRK please post your questions on The MAVRK Toolbox Forum. Please post only comments about the article* ***CC256x TI Bluetooth Stack SPPDemo App*** *here.*	*For technical s please post yo questions at http://e2e.ti.cor Please post on comments abo article* ***CC256x Bluetooth Sta SPPDemo Ap*** }}

Links

Amplifiers & Linear	DLP & MEMS	Processors	Switches & Multiplexers
Audio	High-Reliability	• ARM Processors	Temperature Sensors & Control ICs
Broadband RF/IF & Digital Radio	Interface	• Digital Signal Processors (DSP)	Wireless Connectivity
Clocks & Timers	Logic	• Microcontrollers (MCU)	
Data Converters	Power Management	• OMAP Applications Processors	