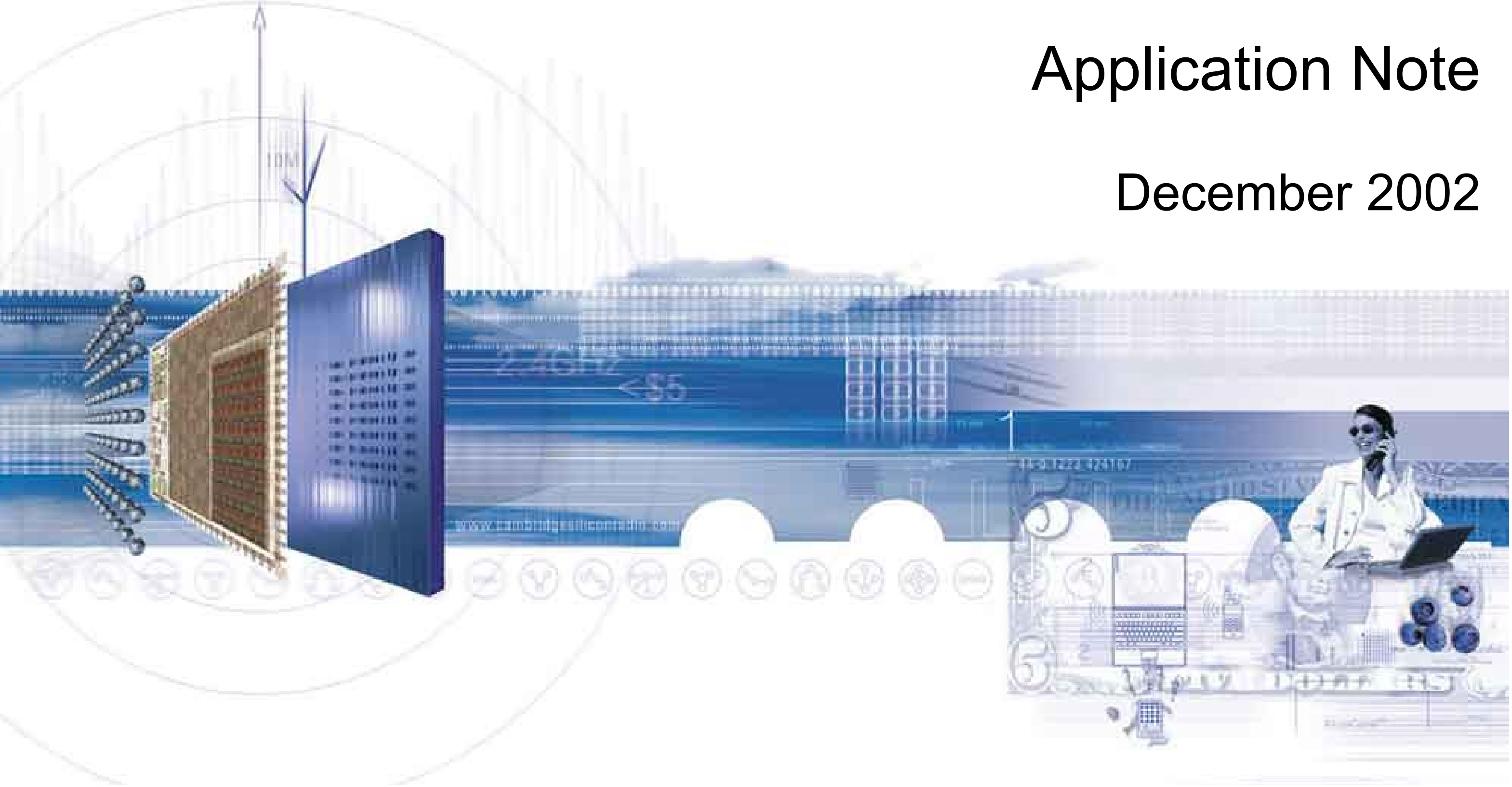
BlueCore™

Accessing Service Discovery Using RFCLI and TCL

Application Note

December 2002



CSR

Unit 400 Cambridge Science Park
Milton Road
Cambridge
CB4 0WH
United Kingdom

Registered in England 3665875

Tel: +44 (0)1223 692000
Fax: +44 (0)1223 692001
www.csr.com

Contents

List of Figures

List of Tables

List of Equations

1 Introduction

The purpose of this application note is to explore the service discovery mechanism available in the Bluetooth™ Specification v1.1. The application note provides an understanding of the service discovery database (SDD) and the service discovery protocol (SDP) used to access it, through the use of RFCLI and test tool command language (TCL).

Using **BlueCore™** devices running an RFCOMM firmware stack based on BlueStack™, the application note explores aspects of SDP that allow applications to discover services that are available on other Bluetooth devices alongside the properties of those services.

This application note builds on the information contained within the Accessing RFCOMM Using RFCLI and TCL Application Note (bcore-an-006Pa), uses the knowledge gained here, and therefore is a follow on to that application note.

This application note works through the following to give a thorough understanding of what SDD and SDP are and how to use them:

- The theory behind SDD, SDP and services
- Example of how to:
  - Access services on a Bluetooth profile e.g. Headset
  - Put a service into the SDD
  - Use the local device to read services on the remote device database

2 Service Discovery

The service discovery operation is based on a client/server architecture shown in Figure 2.1. The architecture is a request/response model between a SDP client running on one Bluetooth™ device and a SDP server running on another Bluetooth device. The actual service discovery procedure permits the client application to discover the presence of services on a server application including the attributes of these services.

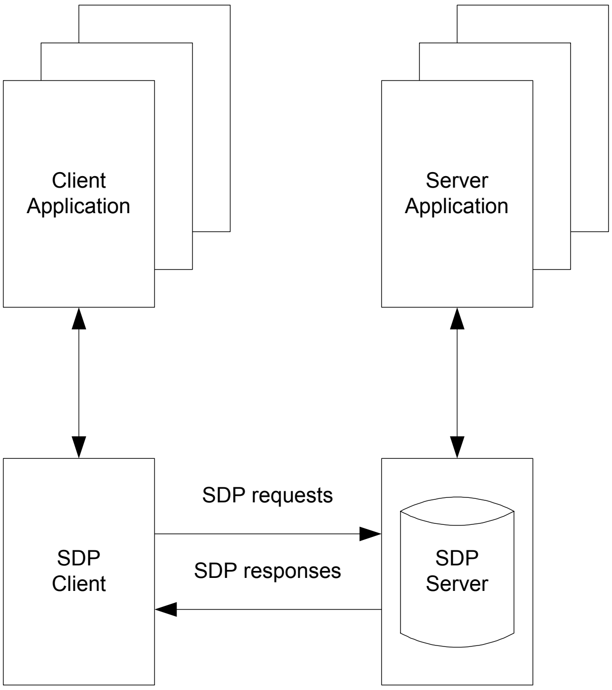


Figure 2.1: Service Discovery Client/Server Architecture

This section covers the theory of the SDD and the SDP and the various attributes and properties of the assorted elements that make up the SDP. The topics covered include:

- The SDP is the protocol that is responsible for the communication between a service discovery server and a client.
- The server maintains a list of service records that describe the properties of services associated with the server
- Services have services classes associated with them
- Data is passed via the SDP requests and responses using protocol data units (PDU)
- The method for discovering services and their attributes

A Bluetooth device can be a client, or a server or both at the same time. If a device contains multiple applications running on it then the SDP server is responsible for advertising all these services to any client that that like to use them. The similar situation is true of the device that contains multiple client applications; which are responsible for enquiring for services on the SDP server on another device on behalf of the client applications it is running.

The requirements placed on the underlying transport layer by SDP are minimal leading to a simple protocol that can function over a reliable or unreliable packet transport layer. Although to function over an unreliable packet transport layer the client would require an implementation that includes timeouts and retransmissions.

2.1 Protocol Data Unit

Figure 2.1 outlines a client/server architecture that uses a request/response model to communicate between the client and the server. The PDU is the communication packet that is passed in either direction between the client and server. A transaction between the client and the server consists of a pair of PDUs, one for the request and, one for the response. In general the client requests are allowed to be pipelined and the server responses can be returned out of order. In the specific case where the SDP uses the L2CAP layer, multiple PDUs may be merged into a single L2CAP packet. Although in the case of when the L2CAP layer is utilised only one SDP request may be outstanding per connection. This means that a client must receive a response to each request before issuing its next request on the same L2CAP connection. By limiting the number of SDP requests permitted to just one unacknowledged packet affords a simpler form of flow control to be implemented.

The format of the PDU is shown in Figure 2.2, a PDU consists of a header field and a parameters field. The header has three elements, which are the PDU ID, the Transaction ID, and the Parameter Length.

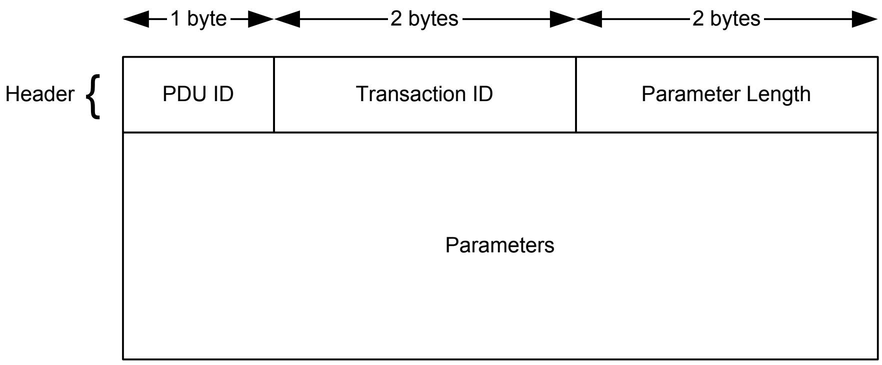


Figure 2.2: The PDU Format

The PDU ID is a 1byte value in the header block that identifies the type of PDU. It defines the meaning and the parameters as defined in Table 2.1.

PDU ID Value	Parameter Description
0x00	Reserved
0x01	SDP_ErrorResponse
0x02	SDP_ServiceSearchRequest
0x03	SDP_ServiceSearchResponse
0x04	SDP_ServiceAttributeRequest
0x05	SDP_ServiceAttributeResponse
0x06	SDP_ServiceSearchAttributeRequest
0x07	SDP_ServiceSearchAttributeResponse
0x07-0xff	Reserved

Table 2.1: PDU ID Values

The transaction ID is used to give a unique identity to a request PDU. This transaction ID has to be replicated in the response PDU in order to link the response with its corresponding request. The transaction ID is a 2byte value that is set by the client, this can be set to any value as long as the value differs from that of any outstanding requests.

The parameter length field in the header is a 2byte value that represents the length in bytes of all parameters contained within the PDU.

2.1.1 Continuation State Parameter

A server is permitted to send a partial response to a client, this occurs when the response being sent is larger than the size of a single response PDU. In this case the server is allowed to split the response over more than one response PDU. In order to achieve this the PDU has a continuation state parameter; this is shown in Figure 2.3 as the PDU with the continuation state parameter appended to it.

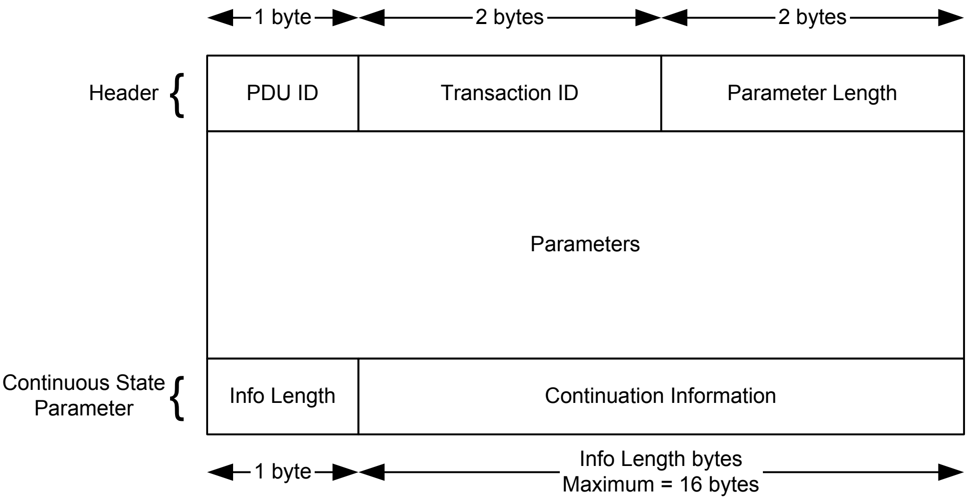


Figure 2.3: PDU with Continuation State Parameter.

The continuous state parameter is available in all PDU responses, but in the majority of cases the responses fit within a single PDU and therefore the continuous state is a single byte that is set to zero. When the server requires splitting the response across PDUs it makes use of the Info Length field and the Continuation Information field inside the continuation state parameter. The Info Length field is a 1byte value that states the number of bytes that make up the Continuation information and therefore the over all continuation state parameter is of variable length but restricted by the Info Length field that is only allowed to take a maximum value of 0x10 hex.

The information that is contained within the Continuation Field does not have a standard format and differs amongst servers. This means that the continuation state parameter generated by a server is relevant only to that server. Also the server is permitted to make the split of the response anywhere it chooses. For the client to receive the complete response it needs to first receive the initial response with the continuation state parameter set. The client will then re-issue its original request but with a different transaction ID and include the continuation state in this new request indicating the client's desire to receive the rest of the original response.

2.1.2 Error PDU

In Section 2.1 transactions between a client and a server were described in terms of a request/response model. The PDUs are grouped in pairs consisting of a request PDU and a response PDU. Normally for each request PDU there is a response PDU corresponding to it, these are outlined in Table 2.1. For example, the SDP_ServiceSearchRequest is a request PDU that searches for a service on the server and it has a corresponding response PDU called SDP_ServiceSearchResponse that returns the appropriate data to the client with respect to its request.

If for any reason the server cannot respond correctly to the client with an appropriate PDU type or that the request from the client has been incorrectly formatted then the server will respond with the error PDU. The error PDU is the PDU ID value of 0x01 hex shown in Table 2.1 and is called SDP_ErrorResponse.

The error PDU has the same structure as the PDU format in Figure 2.2, the PDU ID is as discussed previously as SDP_ErrorResponse in Section 2.1, the transaction ID is the transaction ID of the client request PDU that is in error and the parameters associated with the error PDU are ErrorCode and ErrorInfo.

The ErrorCode is a 2byte value that takes the values listed in Table 2.2 and represents the reason for error PDU being generated. The ErrorInfo parameter is specifically related to a given ErrorCode. At present the current ErrorCodes defined in the Bluetooth Specification v1.1 do not specify the format of any ErrorInfo field.

Value	Parameter Description
0x0000	Reserved
0x0001	Invalid or unsupported SDP version
0x0002	Invalid service record handle
0x0003	Invalid request syntax
0x0004	Invalid PDU size
0x0005	Invalid continuation state
0x0006	Insufficient resources to satisfy request
0x0007-0xffff	Reserved

Table 2.2: Table of Error Codes Associated with SDP_ErrorResponse

2.2 SDP Services

The SDD is a database that exists on a server and contains a set of records responsible for characterising all the services available on a Bluetooth device. The SDP is the protocol that is used to look at these services. The services are used by a SDP client from the information on the services supplied by the server. The information on services that are exchanged between a client and a server are packaged within service records. This section describes the properties and the format of these services.

2.2.1 Service Class

A service record is described in Section 2.2 and Section 2.2.2 as a package of information that describes a service. A service record contains a list of service classes that a specific service may adhere to; an example of a list of these service classes can be seen in Table 2.3 that represents a service record for a Bluetooth headset as shown in Section 5.3 of the Headset Profile in the Bluetooth Specification v1.1. The service class list is shown under the ServiceClassIDList parameter.

Item	Type	Value	Attribute ID
ServiceRecordHandle	uint32	Assigned by server	0x0000
ServiceClassIDList			0x0001
ServiceClass0	UUID	Headset	0x1108
ServiceClass1	UUID	Generic audio	0x1203
ProtocolDescriptorList			0x0004
Protocol0	UUID	L2CAP	0x0100
Protocol1	UUID	RFCOMM	0x0003
ProtocolSpecificParameter0	uint8	Server channel number	
BluetoothProfileDescriptorList			0x0009
Profile0	UUID	Headset	0x1108
Parameter0	uint16	Version 1.0	0x0100
ServiceName	string	“Headset”	0x0000 + language offset
Remote Audio Volume Control	boolean	false	0x0302

Table 2.3: Bluetooth Headset Service Record

Each service belongs to a service class, the service class defines all the attributes of the service by ID, intended use and format of the attribute value. Figure 2.4 outlines the relationship between the service class and the service record. In Figure 2.4 the ServiceClassIDList is shown split into a Class ID know as the super class and the Sub Class ID. The sub class contains additional attribute definitions that are specific to the subclass, which means that it inherits the attributes from a super class and defines more specific ones. Each service class is a subclass of another class whose identifier is contained within the ServiceClassIDList. Each service class within this list is assigned a universally unique identifier (UUID) to identify the service class, for further details see Section 2.2.4.

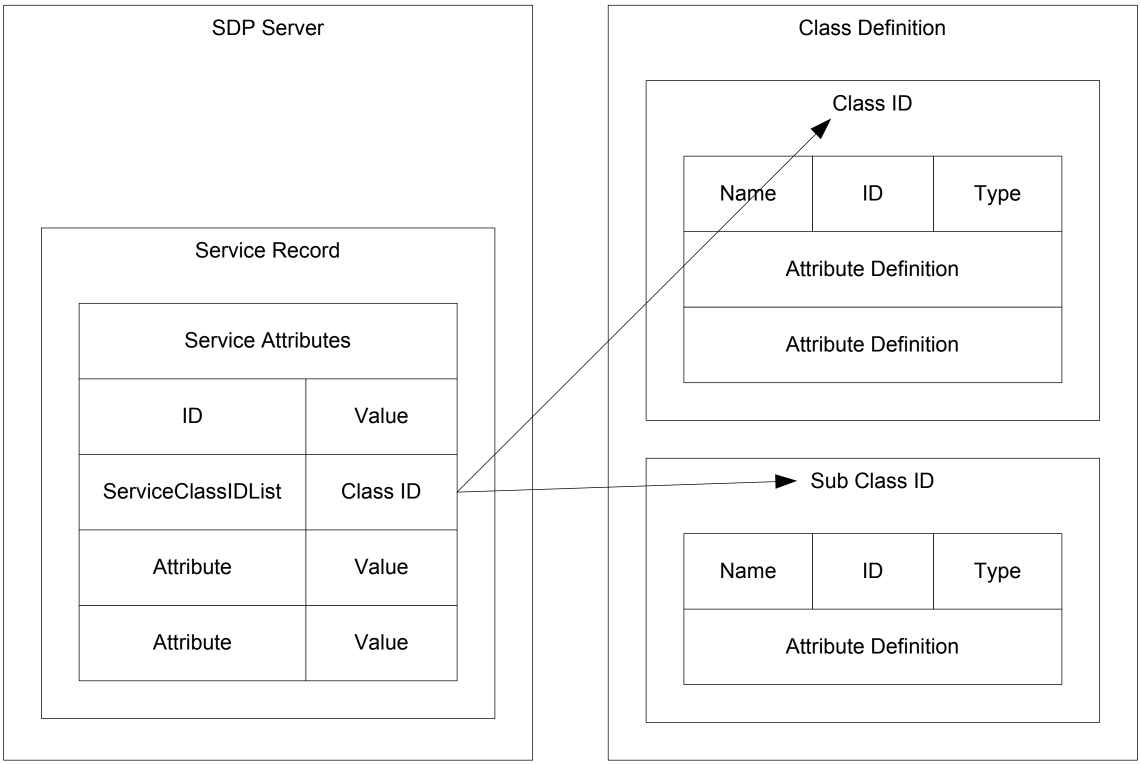


Figure 2.4: SDP Service Classes

2.2.2 Service Record

A service record provides information about a service, whether this is hardware, software, or a combination of hardware and software. In other words a service record advertises a device's capabilities but does not control the resource or perform actions for it. All information about a service is contained and maintained within a single service record. The basic structure of a service record is a catalogue that lists all the service attributes, for further information on service attributes see Section 2.2.3.

A service record position within the SDP server can be located by the use of a unique identifier, this is a 32-bit number called the service record handle. The service record handle is unique to a specific server, so even if two servers contain the same service records, the service records that exist on the different SDP servers are independent identifiers. There is though one service record handle that is the same throughout all SDP servers and this is the service record for the SDP server itself, it has a 32-bit value of 0x00000000 (hex). The zero value service record handle is responsible for holding information on the attributes of the server and the protocol it supports. An example of one of these attributes is the list of the versions of the protocol, which the SDP server supports.

Whilst an L2CAP connection is established the SDP server must ensure that if it removes a service record then it must not reuse the service record handle. Otherwise if a client attempts to use this service record handle an SDP_ErrorResponse will be returned from the server with an ErrorCode of 0x0002 (hex) representing the fact that the client supplied an invalid service record. The service record handle value can be reused once the L2CAP link has be disconnected and restarted. If a service record is removed or added the SDP has no mechanism to automatically notify the clients of any change.

2.2.3 Service Attribute

As mentioned in Section 2.2.2 a service record contains a list of service attributes as shown Figure 2.5. A service attribute is a service description that describes through a range of data types the following:

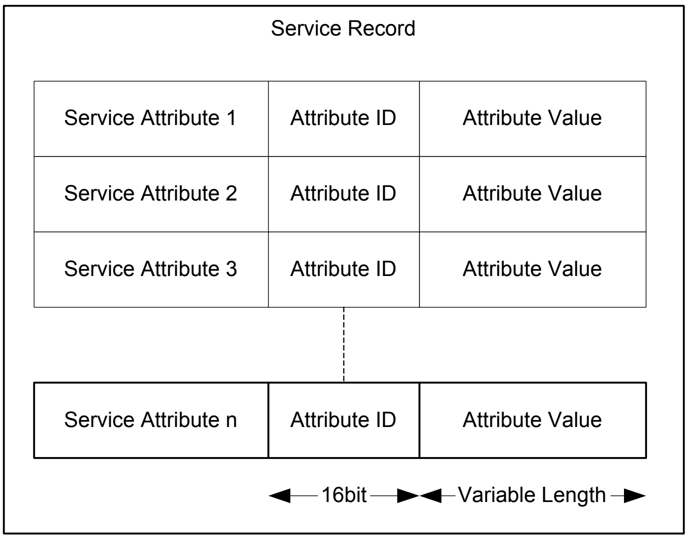
- The supported service type
- Service ID
- Protocols supported
- Service

As shown in Figure 2.5 the service attribute consists of two elements, the attribute ID, and the attribute value. The attribute ID is a 16-bit unsigned integer value that distinguishes each service attribute from other service attributes in the service record. The attribute value is a variable sized field whose meaning is determined by the attribute ID and the service class of the service record associated with it. The attribute value is represented by a data element and in general any type of data element is permitted, see Section 2.2.5 for further details on data elements. The constraints on the data elements used in a service attribute are specified by the service class definition. Only the service classes that are directly supported by the SDP server have their service attribute definitions listed in the Bluetooth Specification v1.1. There are three service attribute definitions groups defined and these can be found on p.366 of Section 5 of the Service Discovery Protocol. The three service attribute definition groups are:

1. The universal attribute definitions which are:
    - ServiceRecordHandle
    - ServiceClassIDList
    - ServiceRecordState
    - ServiceID
    - ProtocolDescriptorList
    - BrowseGroupList
    - LanguageBaseAttributeIDList
    - ServiceInfoTimeToLive
    - ServiceAvailability
    - BluetoothProfileDescriptorList
    - DocumentationURL
    - ClientExecutableURL
    - IconURL
    - ServiceName
    - ServiceDescription
    - ProviderName
    - Reserved universal attribute IDs are in the range 0x000d – 0x01ff (hex)
2. Service discovery server attribute definitions which are:
    - ServiceRecordHandle
    - ServiceClassIDList
    - VersionNumberList
    - ServiceDatabaseState
    - Reserved service discovery server attribute IDs are in the range 0x0202-0x02ff (hex)

3. Browse group descriptor attribute definitions which are:
   - ServiceClassIDList
   - GroupID
   - Reserved browse group descriptor attribute IDs are in the range 0x0201-0x02ff (hex)

With respect to service attributes there are only two attributes that are compulsory in every service record, these are the ServiceRecordHandle and the ServiceClassIDList.



Service Record

Service Attribute 1	Attribute ID	Attribute Value
Service Attribute 2	Attribute ID	Attribute Value
Service Attribute 3	Attribute ID	Attribute Value
Service Attribute n	Attribute ID	Attribute Value

Figure 2.5: Service Attributes Within a Service Record

2.2.4 UUID

In Section 2.2.1 the UUID was first introduced, the identifier is guaranteed to be unique at any time or in any place. The UUID is a 128-bit value with a format that has been set out by the International Organization for Standardization in ISO 11578:1996.

In order to get around the difficulties of handling a 128-bit UUID value Bluetooth has been allocated a base UUID value. Each individual Bluetooth UUID can then be represented as a 16-bit or a 32-bit value that represents an offset value from the Bluetooth base value.

The Bluetooth base UUID value is 00000000-0000-1000-8000-00805f9b34fb as set out in the Bluetooth Assigned numbers document, as mentioned earlier a service can be allocated either a 16-bit or a 32-bit value this is known as an alias. To calculate the full 128-bit UUID from the alias value the two arithmetic operations shown in Equation 2.1 and Equation 2.2 below are used.

$$\text{128bit UUID} = \text{16-bit Alias Value} \times 2^{96} + \text{Bluetooth Base UUID Value}$$

Equation 2.1: 128-bit UUID Calculation Using 16-bit Bluetooth Alias UUID Value

$$\text{128bit UUID} = \text{32-bit Alias Value} \times 2^{96} + \text{Bluetooth Base UUID Value}$$

Equation 2.2: 128-bit UUID Calculation Using 32-bit Bluetooth Alias UUID Value

2.2.5 Data Representation

Section 2.2.3 discussed the use of attributes, which are used to form the service records. The service record consisted of an attribute ID and an attribute value. The attribute value in Figure 2.5 was shown to have a variable length field, in order that a device receiving an attribute understands what size and type of attribute is being sent to it, the first byte of data in the attribute value represents a data element descriptor for the subsequent data contained in the attribute value. Apart from the first byte of data all other data contained within the attribute value are known as data elements.

The data element descriptor is shown in Figure 2.6 where it consists of a 5-bit type field in the most significant bits and a 3-bit size field in the least significant bits.

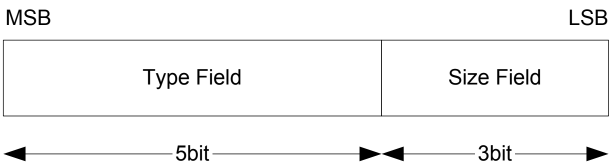


Figure 2.6: Data Element Descriptor

The data element descriptor are shown in Table 2.4, there are nine types that are used to describe the type of attribute contained within the remaining data elements of the attribute value.

5-bit Value	Data Element Type
0	The null type
1	Unsigned integer
2	Signed 2's complement integer
3	UUID
4	Text String
5	Boolean
6	A list of data elements
7	A list of alternative data elements from which one data element is to be selected from.
8	Uniform resource locator (URL)

Table 2.4: Data Element Type Field

The data element descriptor size field shown in Figure 2.6 represents the size of the attribute in the data element. Table 2.5 lists the available sizes of a data element.

3-bit Value	Data Element Size
0	1 byte, or 0 bytes if null type specified in the data element type
1	2 bytes
2	4 bytes
3	8 bytes
4	16 bytes
5	Number of bytes specified in the next 1 byte
6	Number of bytes specified in the next 2 bytes
7	Number of bytes specified in the next 4 byte

Table 2.5: Data Element Size Field

Examples of how to use the type and size fields are shown in Figure 2.7 and Figure 2.8. Figure 2.7 demonstrates how to specify that the data contained within a data element is a 16-bit unsigned integer; and Figure 2.8 is an example of how the text string "CSR" would be formatted in the service attribute.

5bit	3bit	8bit
Type Field = 1	Size Field = 1	Data Byte (LSB)
Data Byte (MSB)		

Figure 2.7: 16-bit Unsigned Integer Data Element

Figure 2.7 has the type field set to 1 denoting the data element is an unsigned integer and with the size field set to 1 it means that the unsigned integer is 2 bytes long i.e. a 16-bit unsigned integer.

5bit	3bit	8bit
Type Field = 4	Size Field = 5	Size = 3bytes
“C”		“S”
“R”		

Figure 2.8: Text Data Element

Figure 2.8 has the type field set to 4 denoting the data element is a text string and with the size field set to 5 it means that the size of the data element is contained in the next byte. The next byte is the text data element, it is three bytes long i.e. three text bytes to allow for the “CSR” string.

2.3 Searching and Browsing for Services

The SDP client can find the services that are contained within a SDP server by the use of searching and browsing. The difference between searching and browsing is that browsing uses a search mechanism to look through the services that the SDP server has to offer. Normally a SDP client searches for services based on properties using UUIDs. The UUID is passed as a search pattern to the server from the client and a match of this UUID is requested.

If the client needs to gain information about a service from the server without any prior knowledge of the services contained within the server’s database, then it uses the browse mechanism to scan the hierarchy of the server database. In order for the server’s database to allow browsing it uses a structure that is based on an attribute that is shared by all service classes called the BrowseGroupList attribute. The BrowseGroupList attribute is a list of UUIDs, where each UUID represents a browse group that a service is linked just in terms of browsing.

Note:

> At present the use of browsing has not been truly adopted in any of the existing Bluetooth profiles. Profiles at present seem to be based on a single or multiple collection of simple service records and does not require the ability to browse for these records. This does not preclude in the future that this feature will not be implemented in new profiles this explains the rather brief discussion above.

3 SDP API Primitives

This section describes the APIs available within BlueStack that can be used to register service records and perform service discovery. Initially the functionality of the individual APIs that are associated with the primitives for the client and server are described, and then these APIs are used to demonstrate:

- Registering the service with the database, see Section 3.1.2
- Examining how to access the services on a device supporting the Headset profile, see Section 3.4

3.1 SDP Server API Primitives

This section describes the API primitives that are available to the server. It gives a summary of the individual primitives and the TCL written library procedures with worked examples where appropriate. For a thorough description of the individual APIs for the server primitives for the SDP refer to the BlueStack User Manual. In comparison to the client functions there are distinctly less API primitives associated with the server, essentially the server primitives are associated with the registration of services and the configuration of the server.

3.1.1 SDS_CONFIG_REQ

The `SDS_CONFIG_REQ` primitive allows the L2CAP MTU size to be specified for all SDS sessions. The parameters supplied are:

- `mtu`, this is the L2CAP MTU size advertised by the device(1)

3.1.2 SDS_REGISTER_REQ

The `SDS_REGISTER_REQ` primitive requests the server to register a service, which is then available to other Bluetooth devices, the parameters supplied are:

- `phandle`, protocol handle to the higher entity using SDP(2)
- `num_rec_bytes`, the number of bytes in a service record
- `*service_rec`, the list of attribute IDs and values that make up the service record

An example of how to register a service record with the server is shown in the TCL script below that can be run in RFCLI. The script represents registering the service record for the Audio Gateway profile:

Note:

(1) Default val used if val supplied is invalid

(2) If the application is running off chip, then `phandle` has to have the top bit set. If the application is running on chip, then `phandle` has to have the top bit clear:

- Off Chip : phandle >= 8000
- On Chip : phandle <= 7fff

BlueCore™ Accessing Service Discovery Using RFCLI and TCL

```
# Register Audio Gateway Service Record

proc register_ag_profile_service_record {ag_server_chan} {
  set ag_record {0x09 0x00 0x01 \
                 0x35 0x06 \
                 0x19 0x11 0x12 \
                 0x19 0x12 0x03 \
                 0x09 0x00 0x04 \
                 0x35 0x0c \
                 0x35 0x03 \
                 0x19 0x01 0x00 \
                 0x35 0x05 \
                 0x19 0x00 0x03 \
                 0x08}
  set ag_record [concat $ag_record $ag_server_chan]
  set ag_record [concat $ag_record {0x09 0x00 0x09 \
                                    0x35 0x08 \
                                    0x35 0x06 \
                                    0x19 0x11 0x12 \
                                    0x09 0x01 0x00 \
                                    0x09 0x01 0x00 \
                                    0x25 0x0d \
                                    0x56 0x6f 0x69 \
                                    0x63 0x65 0x20 \
                                    0x47 0x61 0x74 \
                                    0x65 0x77 0x61 0x79}]

  SDS_REGISTER_REQ 0x8000 $ag_record [llength $ag_record]
  SDS_REGISTER_CFM
}

BC_connect com2 bcsp 115200
# store the server_chan away just in case gets overwritten by the flow ctrl
RFC_DATA_INDs
set agemu_server_chan $server_chan

# Register the sdp record
register_ag_profile_service_record $agemu_server_chan
```

A typical output from RFCLI would be as follows:

```
---- 11:42:27.116 ------------------
SDS_REGISTER_REQ_T
        type =  0e
        phandle =  8000
        num_rec_bytes =  3b
09 00 01 35 06 19 11 12 19 12 03 09 00 04 35 0c 35 03 19 01 00 35 05 19 00
03 08 01 09 00 09 35 08 35 06 19 11 12 09 01 00 09 01 00 25 0d 56 6f 69 63
65 20 47 61 74 65 77 61 79

---- 11:42:27.132 ------------------
SDS_REGISTER_CFM_T
        type =  0f
        phandle =  8000
        svc_rec_hndl =  10000
        result =  00
```

In this example the audio gateway service record as it appears in the Profile document in Table 5.2 on Page 221 of the Bluetooth Specification v1.1 is modified and shown in Table 3.1. The structure of the table is governed by the sequential listing of the above TCL script, this demonstrates how service records can be constructed from their profile specifications.

Item	Definition	Type	Value	Attribute ID	TCL Script
ServiceClassIDList		uint16		0x0001	`0x09 0x00 0x01`
	Defines the list data type and size	data element of 6 bytes made up of 2 UUIDs			`0x35 0x06`
ServiceClass0		UUID	Headset Audio Gateway		`0x19 0x11 0x12` |————| UUID
ServiceClass1		UUID	Generic Audio		`0x19 0x12 0x03` |————| UUID
ProtocolDescriptorList		uint16		0x0004	`0x09 0x00 0x04`
	Defines the list data type and size	data element of 12 bytes made up of 2 data element lists, with 3 bytes and 5 bytes respectively			`0x35 0x0c`
	Defines the list data type and size	data element of 3 bytes made up of UUID			`0x35 0x03`
Protocol0		UUID	L2CAP	0x0100	`0x19 0x01 0x00`
	Defines the list data type and size	Data element of 5 bytes made up of UUID and 1 byte			`0x35 0x05`
Protocol1		UUID	RFCOMM	0x0003	`0x19 0x00 0x03`

Protocol Specific Parameter0	Server Channel	uint8	Server channel number		`0x08`
					`set ag_record [concat $ag_record $ag_server_chan]`

BluetoothProfileDescriptorList		uint16		0x0009	`set ag_record [concat $ag_record {0x09 0x00 0x09}`
	Defines the list data type and size	Data element of 8 bytes made up of another list of 6 bytes			`0x35 0x08`
	Defines the list data type and size	Data element of 6 bytes			`0x35 0x06`
Profile0	Supported profile	UUID	Headset		`0x19 0x11 0x12`

Item	Definition	Type	Value	Attribute ID	TCL Script
Param0	Profile version	uint16	0x0100 (v1.0)		0x09 0x01 0x00
ServiceName				0x0000 + offset of 0x100	0x09 0x01 0x00
	Displayable Text Name	string of 13 bytes	Service-provider defined		0x25 0x0d 86 111 105 99 101 32 71 97 116 101 119 97 121 = "Voice Gateway"

Table 3.1: Audio Gateway Profile with Associated TCL Script

3.1.3 SDS_REGISTER_CFM

The SDS_REGISTER_CFM primitive is returned in response to the SDS_REGISTER_REQ in Section 3.1.2. The parameters returned are:

- phandle, protocol handle to the higher entity using SDP
- result, response of 0x0000 indicates success any other value indicates fail[1]
- svc_rec_hndl, is a 32-bit value with a range of 0x0001000-0xffffffff uniquely identifying each service record on a particular server

An example of the use of this primitive is shown complementing the SDS_REGISTER_REQ primitive used in Section 3.1.2. The output from this primitive shown in RFCLI is:

```
---- 11:42:27.132 ------------------
SDS_REGISTER_CFM_T
        type =  0f
        phandle =  8000
        svc_rec_hndl =  10000
        result =  00
```

The return parameters show that the service record was registered successfully; this is denoted by the value of result being 0x00. The service record handler is 0x00010000.

Note:

[1] Reference file **sds_prim.h** for other result codes

3.1.4 SDS_UNREGISTER_REQ

The SDS_UNREGISTER_REQ primitive has the opposite effect to the primitive SDS_REGISTER_REQ in Section 3.1.2. It requests that a previously registered service be unregistered from the server. It requires the service record handle value to be able to remove it. The parameters that need to be supplied are:

- phandle, protocol handle to the higher entity using SDP
- svc_rec_hndl, is a 32-bit value with a range of 0x0001000-0xffffffff uniquely identifying each service record on a particular server

An example of how to use the primitive is shown below as a TCL script:

```
# Register Audio Gateway Service Record

proc register_ag_profile_service_record {ag_server_chan} {

  set ag_record {0x09 0x00 0x01 \
                 0x35 0x06 \
                 0x19 0x11 0x12 \
                 0x19 0x12 0x03 \
                 0x09 0x00 0x04 \
                 0x35 0x0c \
                 0x35 0x03 \
                 0x19 0x01 0x00 \
                 0x35 0x05 \
                 0x19 0x00 0x03 \
                 0x08}
  set ag_record [concat $ag_record $ag_server_chan]
  set ag_record [concat $ag_record {0x09 0x00 0x09 \
                                    0x35 0x08 \
                                    0x35 0x06 \
                                    0x19 0x11 0x12 \
                                    0x09 0x01 0x00 \
                                    0x09 0x01 0x00 \
                                    0x25 0x0d \
                                    0x56 0x6f 0x69 \
                                    0x63 0x65 0x20 \
                                    0x47 0x61 0x74 \
                                    0x65 0x77 0x61 0x79}]

  SDS_REGISTER_REQ 0x8000 $ag_record [llength $ag_record]
  set result [SDS_REGISTER_CFM]
  set service_record_handle [lindex $result 1]
  set retval [lindex $result 2]
  puts "Service Record Handle: $service_record_handle"
  set retval [concat $retval $service_record_handle]
  puts "retval: $retval"
  return $retval
}

BC_connect com2 bcsp 115200
# store the server_chan away just in case gets overwritten by the flow ctrl
RFC_DATA_INDs
set agemu_server_chan $server_chan

# Register the sdp record
set result [register_ag_profile_service_record $agemu_server_chan]
if {[lindex $result 0]} {
  puts "SDS_REGISTER_REQ unsuccessful"
  puts "Result Code: [lindex $result 0]"
} else {
  puts "SDS_REGISTER_REQ successful"
  puts "Unregister Service with service record handle: [lindex $result 1]"
  SDS_UNREGISTER_REQ 0x8000 [lindex $result 1]
  SDS_UNREGISTER_CFM
}
```

This script is an extension of the example used in Section 3.1.2. The basic concept of this script is to, register a service and store its service record handle; then use this service record handle to be able to remove this service record from the server. The RFCLI output expected for this script is shown below and this confirms that the service was first registered with the service record handle set to 0x00010000 and this handle value was passed to the `SDS_UNREGISTER_REQ` primitive and the resulting `SDS_UNREGISTER_CFM` denotes that the service was unregistered successfully.

```
---- 13:28:19.873 -------------------
SDS_REGISTER_REQ_T
        type =  0e
        phandle =  8000
        num_rec_bytes =  3b
09 00 01 35 06 19 11 12 19 12 03 09 00 04 35 0c 35 03 19 01 00 35 05 19 00
03 08 01 09 00 09 35 08 35 06 19 11 12 09 01 00 09 01 00 25 0d 56 6f 69 63
65 20 47 61 74 65 77 61 79

---- 13:28:19.889 -------------------
SDS_REGISTER_CFM_T
        type =  0f
        phandle =  8000
        svc_rec_hndl =  10000
        result =  00
Service Record Handle: 0x10000
retval: 0x0 0x10000
SDS_REGISTER_REQ successful
Unregister Service with service record handle: 0x10000

---- 13:28:19.936 -------------------
SDS_UNREGISTER_REQ_T
        type =  10
        phandle =  8000
        svc_rec_hndl =  10000

---- 13:28:19.936 -------------------
SDS_UNREGISTER_CFM_T
        type =  11
        phandle =  8000
        svc_rec_hndl =  10000
        result =  00
```

3.1.5 SDS_UNREGISTER_CFM

The SDS_UNREGISTER_CFM primitive is returned in response to the SDS_UNREGISTER_REQ in Section 3.1.4. The parameters returned are:

- phandle, protocol handle to the higher entity using SDP
- result, response of 0x0000 indicates success any other value indicates fail
- svc_rec_hndl, is a 32-bit value with a range of 0x0001000-0xffffffff uniquely identifying each service record on a particular server

An example of the use of this primitive is shown complementing the SDS_REGISTER_REQ primitive used in Section 3.1.4. The output from this primitive shown in RFCLI is:

```
---- 13:28:19.936 -------------------
SDS_UNREGISTER_CFM_T
        type =  11
        phandle =  8000
        svc_rec_hndl =  10000
        result =  00
```

The return parameters show that the service record was unregistered successfully, denoted by the value of result being 0x00 and the service record handler unregistered was 0x00010000.

3.1.6 sds_register_req

`sds_register_req` is a library function that calls `SDS_REGISTER_REQ` and then waits for `SDS_REGISTER_CFM`. The parameters that are passed to the routine are the same as `SDS_REGISTER_REQ` in Section 3.1.2 and the returned values are the same as the `SDS_REGISTER_CFM` in Section 3.1.3.

An example of how to use the sds_register_req procedure is to modify the example in Section 3.1.2 to the script below. Here the two individual primitives SDS_REGISTER_REQ and SDS_REGISTER_CFM have been replaced with the sds_register_req:

```
# Register Audio Gateway Service Record

proc register_ag_profile_service_record {ag_server_chan} {

  set ag_record {0x09 0x00 0x01 \
                 0x35 0x06 \
                 0x19 0x11 0x12 \
                 0x19 0x12 0x03 \
                 0x09 0x00 0x04 \
                 0x35 0x0c \
                 0x35 0x03 \
                 0x19 0x01 0x00 \
                 0x35 0x05 \
                 0x19 0x00 0x03 \
                 0x08}
  set ag_record [concat $ag_record $ag_server_chan]
  set ag_record [concat $ag_record {0x09 0x00 0x09 \
                                    0x35 0x08 \
                                    0x35 0x06 \
                                    0x19 0x11 0x12 \
                                    0x09 0x01 0x00 \
                                    0x09 0x01 0x00 \
                                    0x25 0x0d \
                                    0x56 0x6f 0x69 \
                                    0x63 0x65 0x20 \
                                    0x47 0x61 0x74 \
                                    0x65 0x77 0x61 0x79}]

  sds_register_req 0x8000 $ag_record [llength $ag_record]
}
```

3.1.7 sds_unregister_req

sds_unregister_req is a library function that calls SDS_UNREGISTER_REQ and then waits for SDS_UNREGISTER_CFM. The parameters that are passed to the routine are the same as SDS_UNREGISTER_REQ in Section 3.1.4 and the returned values are the same as the SDS_REGISTER_CFM in Section 3.1.3.

An example of how to use the sds_unregister_req procedure is to modify the example in Section 3.1.4 to give the script below. Here the two individual primitives SDS_UNREGISTER_REQ and SDS_UNREGISTER_CFM have been replaced with the sds_unregister_req:

```
# Register Audio Gateway Service Record

proc register_ag_profile_service_record {ag_server_chan} {

  set ag_record {0x09 0x00 0x01 \
                 0x35 0x06 \
                 0x19 0x11 0x12 \
                 0x19 0x12 0x03 \
                 0x09 0x00 0x04 \
                 0x35 0x0c \
                 0x35 0x03 \
                 0x19 0x01 0x00 \
                 0x35 0x05 \
                 0x19 0x00 0x03 \
                 0x08}
  set ag_record [concat $ag_record $ag_server_chan]
  set ag_record [concat $ag_record {0x09 0x00 0x09 \
                                    0x35 0x08 \
                                    0x35 0x06 \
                                    0x19 0x11 0x12 \
                                    0x09 0x01 0x00 \
                                    0x09 0x01 0x00 \
                                    0x25 0x0d \
                                    0x56 0x6f 0x69 \
                                    0x63 0x65 0x20 \
                                    0x47 0x61 0x74 \
                                    0x65 0x77 0x61 0x79}]

  SDS_REGISTER_REQ 0x8000 $ag_record [llength $ag_record]
  set result [SDS_REGISTER_CFM]
  set service_record_handle [lindex $result 1]
  set retval [lindex $result 2]
  puts "Service Record Handle: $service_record_handle"
  set retval [concat $retval $service_record_handle]
  puts "retval: $retval"
  return $retval
}

BC_connect com2 bcsp 115200
# store the server_chan away just in case gets overwritten by the flow ctrl
RFC_DATA_INDs
set agemu_server_chan $server_chan

# Register the sdp record
set result [register_ag_profile_service_record $agemu_server_chan]
if {[lindex $result 0]} {
  puts "SDS_REGISTER_REQ unsuccessful"
  puts "Result Code: [lindex $result 0]"
} else {
  puts "SDS_REGISTER_REQ successful"
  puts "Unregister Service with service record handle: [lindex $result 1]"
  sds_unregister_req 0x8000 [lindex $result 1]
}
```

3.2 SDP Client API Primitives

This section describes the API primitives that are available to the client; it gives a summary of the individual primitives and the TCL written library functions. The example TCL script outlined in Section 3.5 covers the majority of these primitives and functions. Table 3.2 summarises where example usage can be located in the example TCL script as well as pointing to the appropriate sections in this application note that document this script. For a thorough description of the individual APIs for the Client primitives for the SDP refer to the BlueStack User Manual.

3.2.1 SDC_CONFIG_REQ

Sets L2CAP MTU size for all SDC sessions; the parameters supplied are:

- `mtu`, L2CAP MTU size, the default value is used if supplied value is invalid

3.2.2 SDC_OPEN_SEARCH_REQ

Requests the opening of a L2CAP connection for multiple searches; the parameters supplied are:

- `bd_addr`, Bluetooth address of the remote device
- `phandle`, protocol handle to the higher entity using SDP

3.2.3 SDC_OPEN_SEARCH_CFM

Returns result of the primitive `SDC_OPEN_SEARCH_REQ`; the parameters supplied are:

- `result`, returns the result of the open search request; valid values are `SDC_OPEN_SEARCH_OK`, `SDC_OPEN_SEARCH_BUSY`, `SDC_OPEN_SEARCH_FAILED`, `SDC_OPEN_SEARCH_OPEN` or `SDC_OPEN_SEARCH_DISCONNECTED`
- `phandle`, protocol handle to the higher entity using SDP

3.2.4 SDC_CLOSE_SEARCH_REQ

Requests closure of a permanent L2CAP connection that had been previously opened with `SDC_OPEN_SEARCH_REQ` primitive, the parameters supplied are:

- `phandle`, protocol handle to the higher entity using SDP

3.2.5 SDC_CLOSE_SEARCH_IND

Indicates that a L2CAP connection opened with a search request has been closed by either the client or the server; the parameters supplied are:

- `result`, returns information on whether disconnection was carried out by the client or server
- `phandle`, protocol handle to the higher entity using SDP

3.2.6 SDC_SERVICE_ATTRIBUTE_REQ

Requests attribute values from a service record on the remote device; the parameters supplied are:

- `*attr_list`, data element list of attribute IDs
- `bd_addr`, Bluetooth address of the remote device
- `max_num_attr`, maximum number of attribute bytes to be returned in one response packet
- `phandle`, protocol handle to the higher entity using SDP
- `svc_rec_hndl`, service record handle being queried
- `size_attr_list`, the size in bytes of the attribute list specified by the `*attr_list`

3.2.7 SDC_SERVICE_ATTRIBUTE_CFM

Returns the resulting attribute values from the search initiated by `SDC_SERVICE_ATTRIBUTE_REQ`; the parameters supplied are:

- `*attr_list`, data element list of attribute IDs
- `err_code`, error code as defined by the SDP_ErrorResponse PDU in the Bluetooth Specification v1.1
- `*err_info`, reserved
- `phandle`, protocol handle to the higher entity using SDP
- `response`, indicates a success (0x0000) or a reason code for failure (any value > 0x0000)
- `size_attr_list`, the size in bytes of the attribute list specified by the `*attr_list`
- `size_err_info`, number of bytes in the error information

3.2.8 SDC_SERVICE_SEARCH_ATTRIBUTE_REQ

Equivalent of the SDP_Service_Search_Attribute_Request in the Bluetooth specification; the parameters supplied are:

- `phandle`, protocol handle to the higher entity using SDP
- `size_attr_list`, the size in bytes of the attribute list specified by the `*attr_list`
- `*attr_list`, data element list of attribute IDs
- `bd_addr`, Bluetooth address of the remote device
- `max_num_attr`, maximum number of attribute bytes to be returned in one response packet

3.2.9 SDC_SERVICE_SEARCH_ATTRIBUTE_CFM

Returns the result of the search request specified by the primitive SDC_SERVICE_SEARCH_ATTRIBUTE_REQ the parameters supplied are:

- phandle, protocol handle to the higher entity using SDP
- total_response_size, total size in bytes of the entire response
- size_attr_list, the size in bytes of the attribute list specified by the *attr_list
- *attr_list, data element list of attribute IDs
- more_to_come, indicates that further responses to come if set to TRUE
- response, indicates a success (0x0000) or a reason code for failure (any value > 0x0000)
- err_code, error code as defined by the SDP_ErrorResponse PDU in the Bluetooth Specification v1.1
- size_err_info, number of bytes in the error information
- *err_info, reserved

3.2.10 SDC_SERVICE_SEARCH_REQ

Equivalent of the SDP_Service_Search_Request PDU in the Bluetooth specification; the parameters supplied are:

- bd_addr, Bluetooth address of the remote device
- max_num_recs, maximum number of record handles to be returned from the search
- phandle, protocol handle to the higher entity using SDP
- *srch_pttrn, search pattern defined as a data element sequence of up to 12 UUIDs
- size_srch_pttrn, number of bytes in the search pattern

3.2.11 SDC_SERVICE_SEARCH_CFM

Returns the result of the search request specified by the primitive SDC_SERVICE_SEARCH_REQ; the parameters supplied are:

- err_code, error code as defined by the SDP_ErrorResponse PDU in the Bluetooth Specification v1.1
- *err_info, reserved
- *rec_list, list of 32-bit handles identifying record that corresponds to the search criteria
- num_recs_ret, number of service records returned
- phandle, protocol handle to the higher entity using SDP
- response, indicates a success (0x0000) or a reason code for failure (any value > 0x0000)
- size_err_info, number of bytes in the error information
- size_rec_list, number of bytes in the list
- size_srch_pttrn, number of bytes in the search pattern
- *srch_pttrn, search pattern defined as a data element sequence of up to 12 UUIDs

3.2.12 SDC_TERMINATE_PRIMITIVE_REQ

Terminates an active service search the parameters supplied are:

- phandle, protocol handle to the higher entity using SDP

3.2.13 sdc_config_req

Configures SDP to use a specific L2CAP MTU when connecting to a server; the parameters supplied are:

- As per `SDC_CONFIG_REQ`, see Section 3.2.1

3.2.14 sdc_open_search

Is a wrapper around `sdc_open_search_req` in Section 3.2.15. The global variable `connectionAttempts` controls how many attempts will be made to open the search before giving up; the parameters supplied are:

- As per `sdc_open_search_req`, see Section 3.2.15

Important Note:

> The use of the library function `sdc_open_search` will cause a TCL script to lock when used with applications that require pairing and authentication. This is due to the fact that the `sdc_open_search` function waits for the primitive `SDC_OPEN_SEARCH_CFM` to be issued before continuing. The headset application with the security level set at two or three, needs to be paired and authenticated before the `SDC_OPEN_SEARCH_REQ` primitive can be confirmed.

3.2.15 sdc_open_search_req

Calls `SDC_OPEN_SEARCH_REQ` and then waits for `SDC_OPEN_SEARCH_CFM`; the parameters supplied are:

- As for `SDC_OPEN_SEARCH_REQ`, see Section 3.2.2

Important Note:

> The use of the library function `sdc_open_search_req` will cause a TCL script to lock when used with applications that require pairing and authentication. This is due to the fact that the `sdc_open_search` function waits for the primitive `SDC_OPEN_SEARCH_CFM` to be issued before continuing. The headset application with the security level set at two or three, needs to be paired and authenticated before the `SDC_OPEN_SEARCH_REQ` primitive can be confirmed.

3.2.16 sdc_close_search_req

Terminates an SDP search session and closes down the ACL link.

3.2.17 sdc_range_search

`sdc_range_search` requests all attributes for a particular service; the parameters supplied are:

- `service`, 16-bit UUID specifying the service

3.2.18 sdc_service_attribute_req

Calls `SDC_SERVICE_ATTRIBUTE_REQ` and then waits for `SDC_SERVICE_ATTRIBUTE_CFM` the parameters supplied are:

- As for `SDC_SERVICE_ATTRIBUTE_REQ`, see Section 3.2.6

3.2.19 sdc_service_search_attribute_req

Calls `SDC_SERVICE_SEARCH_ATTRIBUTE_REQ` and then waits for `SDC_SERVICE_SEARCH_ATTRIBUTE_CFM`; the parameters supplied are:

- As for `SDC_SERVICE_SEARCH_ATTRIBUTE_REQ`, see Section 3.2.8

3.2.20 sdc_service_search_req

Calls SDC_SERVICE_SEARCH_REQ and then waits for SDC_SERVICE_SEARCH_CFM the parameters supplied are:

- As for SDC_SERVICE_SEARCH_REQ, see Section 1.1.1

3.3 Registering a Service Record

Registering and unregistering a service record on the SDP server is adequately covered by the primitives in Section 3.1.2, Section 3.1.3, Section 3.1.4 and Section 3.1.5, as well as the library functions in Section 3.1.6 and Section 3.1.7.

3.4 Headset Service Discovery

In order to understand the way in which the SDP Client searches for attributes and services on the SDP Server a TCL script can be run in an RFCLI session to show how the service records on a true Bluetooth profile can be accessed. In order to run the example script the following set-up is required:

1. The example TCL script is listed in **SDPExample35.tcl**
2. The headset profile available in BlueLab v2.5 is downloaded to the Casira attached to com1 as shown in Figure 3.1. Holding down the Talk, Volume Down and Volume Up buttons together for a few seconds should trigger a hard reset on the headset. This will reset the PIN code and clear the link keys and pairing settings, allowing pairing and authentication to occur.
3. The TCL script in **SDPExample35.tcl** is run in an RFCLI session attached to the Casira on com2 as shown in Figure 3.1. If the parameter `UsingPrimitives` is set to TRUE (1) the script will run using just primitives, but if set to FALSE (0) it will run using as many library functions as possible.
4. The output from running the TCL script successfully is shown in **RFCLI_OutputPrimitives.txt** when `UsingPrimitives` is set to TRUE, and is shown in **RFCLI_OutputLibrary.tcl** when `UsingPrimitives` is set to FALSE.

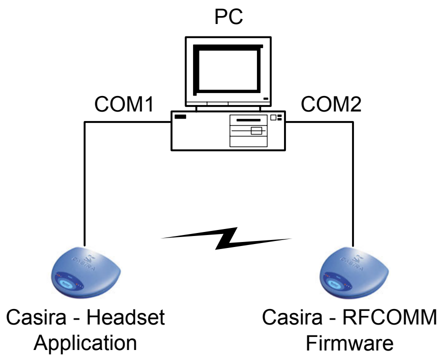


Figure 3.1: Headset Service Discovery Example Set-up

The intention of the example TCL script is to show how to use the SDP API primitives available on BlueStack. Table 3.2 lists where example usage of various primitives and functions can be found within the TCL example in **SDPExample35.tcl**. The example covers all the primitives and functions listed in the Table 3.2, except for the two library functions `sdc_open_search` and `sdc_open_search_req`, whose restrictions are outlined in Section 3.2.14 and Section 3.2.15 respectively. The script has been written to run in two forms:

1. To use the primitives only, this is when `UsingPrimitives` is set to TRUE
2. To use as much of the library functions as possible, this is when `UsingPrimitives` is set to FALSE

The overall structure of the example TCL script is outlined in Figure 3.2 and a description of the code and procedures is covered in Section 3.5.

.

Primitives and Library Functions	Location	Section
SDS_REGISTER_REQ	proc RegisterAGServiceRecord {}	3.5.3
SDS_REGISTER_CFM	proc RegisterAGServiceRecord {}	3.5.3
SDS_UNREGISTER_REQ	proc UnRegisterAGServiceRecord {}	3.5.17
SDS_UNREGISTER_CFM	proc UnRegisterAGServiceRecord {}	3.5.17
sds_register_req	proc RegisterAGServiceRecord {}	3.5.3
sds_unregister_req	proc UnRegisterAGServiceRecord {}	3.5.17
SDC_OPEN_SEARCH_REQ	proc StartSDPSearch {}	3.5.6
SDC_OPEN_SEARCH_CFM	proc StartSDPSearch {}	3.5.10
SDC_CLOSE_SEARCH_REQ	proc mainroutine {}	3.5.18
SDC_CLOSE_SEARCH_IND	proc mainroutine {}	3.5.18
SDC_SERVICE_ATTRIBUTE_CFM	proc ServiceAttributes {}	3.2.7
SDC_SERVICE_ATTRIBUTE_REQ	proc ServiceAttributes {}	3.2.6
SDC_SERVICE_SEARCH_ATTRIBUTE_CFM	proc ServiceSearchAttributes {}	3.5.14
SDC_SERVICE_SEARCH_ATTRIBUTE_REQ	proc ServiceSearchAttributes {}	3.5.14
SDC_SERVICE_SEARCH_CFM	proc SearchHIDServiceRecord {}	3.2.11
SDC_SERVICE_SEARCH_REQ	proc SearchHIDServiceRecord {}	1.1.1
SDC_TERMINATE_PRIMITIVE_REQ	proc TerminateSearch {}	3.2.12
sdc_close_search_req	proc mainroutine {}	3.5.18
sdc_open_search	See Section 3.2.14 for restrictions	
sdc_open_search_req	See Section 3.2.15 for restrictions	
sdc_range_search	proc mainroutine {}	3.5.15
sdc_service_attribute_req	proc ServiceAttributes {}	3.2.18
sdc_service_search_attribute_req	proc ServiceSearchAttributes {}	3.5.14
sdc_service_search_req	proc SearchHIDServiceRecord {}	3.2.20

Table 3.2: Location of Usage of SDP Primitives Within TCL Example Script

3.5 Example TCL Script Structure

Table 3.2 lists all the SDP primitives and function available in RFCLI that interface to the BlueStack SDP API primitives outlined in Section 3.1 and Section 3.2. Table 3.2 also lists the locations in the TCL script shown in **SDPExample35.tcl** of how each of the primitives and functions are implemented. This section documents the functionality of the TCL script shown in **SDPExample35.tcl**. Figure 3.2 shows the overall structure of the script and the subsections 3.5.1 to 3.5.18 highlight the individual procedures that make up the functionality of each functional block shown in Figure 3.2. Within each subsection examples of the SDP primitives and functions being used are highlighted.

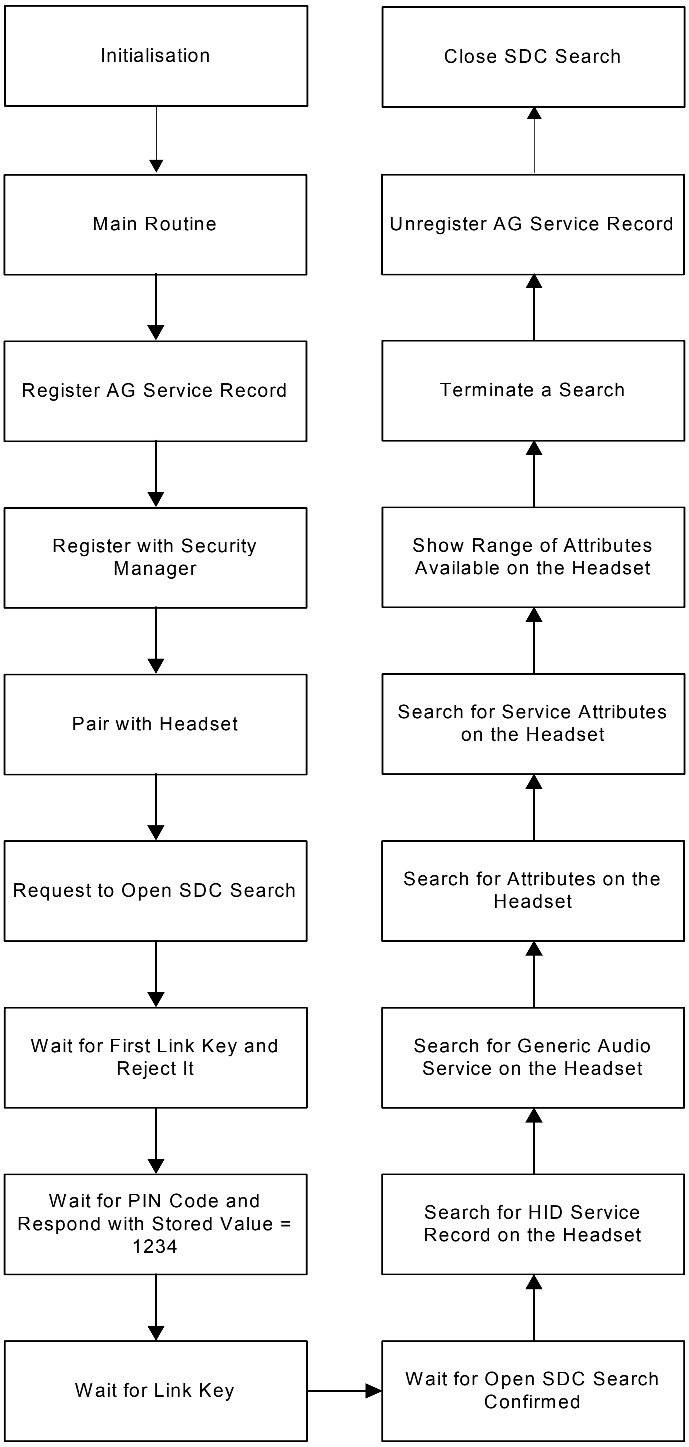

Initialisation
Main Routine
Register AG Service Record
Register with Security Manager
Pair with Headset
Request to Open SDC Search
Wait for First Link Key and Reject It
Wait for PIN Code and Respond with Stored Value = 1234
Wait for Link Key
Wait for Open SDC Search Confirmed
Search for HID Service Record on the Headset
Search for Generic Audio Service on the Headset
Search for Attributes on the Headset
Search for Service Attributes on the Headset
Show Range of Attributes Available on the Headset
Terminate a Search
Unregister AG Service Record
Close SDC Search

Figure 3.2: Structure of TCL Example Script

3.5.1 Initialisation

The script shown below covers initialisation, connecting to the Casira on com2 in Figure 3.1, setting up the Bluetooth address of the target headset and registering with the device manager and RFCOMM layers.

```
# Initialisation
BC_connect com2 bcsp 115200
#Set UsingPrimitives TRUE to use PRIMITIVES, FALSE to use library functions
set UsingPrimitives 1
set bd_addr.lap 0x10e47
set bd_addr.uap 0x5b
set bd_addr.nap 0x02
BC_option -agpmode
# Register with the dm
dm_am_register_req
# Register with the rfcomm
rfc_register_req
# store the server_chan otherwise its overwritten by flow ctrl data inds
set ag_server_chan $server_chan
rfc_init_req
mainroutine
```

The output expected in RFCLI for this script is as follows:

```
---- 11:49:50.884 ------------------
DM_AM_REGISTER_REQ_T
        type =  00
        phandle =  8000

---- 11:49:50.900 ------------------
DM_AM_REGISTER_CFM_T
        type =  01
        phandle =  8000

---- 11:49:50.900 ------------------
RFC_REGISTER_REQ_T
        type =  03
        phandle =  8000

---- 11:49:50.915 ------------------
RFC_REGISTER_CFM_T
        type =  04
        phandle =  8000
        server_chan =  01
        accept =  01

---- 11:49:50.915 ------------------
RFC_INIT_REQ_T
        type =  01
        phandle =  8000
        psm_local =  03
        use_flow_control =  01
        fc_type =  01
        fc_threshold =  03
        fc_timer =  01
        rsvd_4 =  00
        rsvd_5 =  00

---- 11:49:50.931 ------------------
RFC_INIT_CFM_T
        type =  02
        phandle =  8000
        psm_local =  03
        fc_type =  8000
        fc_threshold =  03
        fc_timer =  01
        rsvd_4 =  00
        rsvd_5 =  00
```

3.5.2 Main Routine

The Main Routine in Figure 3.2 is covered by the procedure `mainroutine`. It is the second procedure shown in Figure 3.2 and is responsible for calling all the other TCL procedures shown below it in order. The script for the `mainroutine` is shown below and the comment fields correspond to all the procedures shown in Figure 3.2:

```
# Main Routine
proc mainroutine {} {
  global phandle ag_server_chan UsingPrimitives

  # Register AG Service Record
  set agrecordhandle [lindex [RegisterAGServiceRecord $ag_server_chan] 1]

  # Register with Security Manager
  DM_SM_REGISTER_REQ 1 $ag_server_chan 0 0x13 0

  # Pair with Headset
  PairWithHeadset

  # Request to Open SDC Search
  StartSDPSearch 0

  # Wait for First Link Key and Reject It
  WaitLinkKeyRequestReject

  # Wait for PIN Code and Respond with Stored Value = 1234
  WaitPinCodeRequest

  # Wait for Link Key
  WaitLinkKeyRequest

  # Wait for Open SDC Search Confirmed
  StartSDPSearch 1

  # Search for HID Service Record on the Headset
  SearchHIDServiceRecord

  # Search for Generic Audio Service on the Headset
  SearchGenericAudioServiceRecord

  # Search for Attributes on the Headset
  ServiceAttributes

  # Search for Service Attributes on the Headset
  ServiceSearchAttributes

  # Show Range of Attributes Available on the Headset
  sdc_range_search 0x1108

  # Terminate a Search
  TerminateSearch

  # Unregister AG Service Record
  UnRegisterAGServiceRecord $phandle $agrecordhandle

  # Close SDC Search
  if {$UsingPrimitives} {
    SDC_CLOSE_SEARCH_REQ
    set result [lindex [SDC_CLOSE_SEARCH_IND] 1]
  } else {
    set result [lindex [sdc_close_search_req] 1]
  }
  switch $result {
    0x0 {puts "Connection disconnected by SDS Server"}
    0x1 {puts "Connection disconnected by SDS Client"}
  }
}
```

As mentioned earlier in this subsection the `mainroutine` procedure calls all the other procedures shown in Figure 3.2, therefore the output expected in RFCLI is not shown for this procedure but the output is indicated in the documentation of the subsections that cover the individual procedures called by `mainroutine`.

3.5.3 Register AG Service Record

The Register AG Service Record routine in Figure 3.2 is covered by the procedure `RegisterAGServiceRecord`. This routine demonstrates how to register a service with the SDP server database. As the remote device is a headset, then the local device service record that is registered by this routine is for the audio gateway. The routine returns the service record handle of the registered record via the variable `retval` to the calling routine. The calling routine must supply the server channel value so that this can be inserted in to the record. The script is as follows:

```
# Register AG Service Record
proc RegisterAGServiceRecord {AGServerChannel} {
  global UsingPrimitives
  set hs_record {0x09 0x00 0x01 0x35 0x06 \
                 0x19 0x11 0x12 0x19 0x12 0x03 \
                 0x09 0x00 0x04 0x35 0x0c \
                 0x35 0x03 0x19 0x01 0x00 \
                 0x35 0x05 0x19 0x00 0x03 \
                 0x08}
  set hs_record [concat $hs_record $AGServerChannel]
  set hs_record [concat $hs_record {0x09 0x00 0x09 \
                                    0x35 0x08 \
                                    0x35 0x06 \
                                    0x19 0x11 0x12 \
                                    0x09 0x01 0x00 \
                                    0x09 0x01 0x00 \
                                    0x25 0x0d \
                                    0x56 0x6f 0x69 \
                                    0x63 0x65 0x20 \
                                    0x47 0x61 0x74 \
                                    0x65 0x77 0x61 0x79}]
  if {$UsingPrimitives} {
    SDS_REGISTER_REQ 0x8000 $hs_record [llength $hs_record]
    set result [SDS_REGISTER_CFM]
  } else {
    set result [sds_register_req - $hs_record [llength $hs_record]]
  }
  set service_record_handle [lindex $result 1]
  set retval [lindex $result 2]
  puts "Service Record Handle: $service_record_handle"
  set retval [concat $retval $service_record_handle]
  puts "retval: $retval"
  return $retval
}
```

The output expected in RFCLI for this script is as follows:

```
---- 11:49:50.947 ------------------
SDS_REGISTER_REQ_T
        type =  0e
        phandle =  8000
        num_rec_bytes =  3b
09 00 01 35 06 19 11 12 19 12 03 09 00 04 35 0c 35 03 19 01 00 35 05 19 00
03 08 01 09 00 09 35 08 35 06 19 11 12 09 01 00 09 01 00 25 0d 56 6f 69 63
65 20 47 61 74 65 77 61 79

---- 11:49:50.962 ------------------
SDS_REGISTER_CFM_T
        type =  0f
        phandle =  8000
        svc_rec_hndl =  10000
        result =  00
Service Record Handle: 0x10000
retval: 0x0 0x10000
```

3.5.4 Register with Security Manager

The Register with Security Manager routine in Figure 3.2 is strictly not an individual procedure. The following line of TCL script that appears in the `mainroutine` procedure initiates the registration. Registration with the security manager is performed so that pairing and authentication can be carried out:

```
# Register with Security Manager
DM_SM_REGISTER_REQ 1 $ag_server_chan 0 0x13 0
```

The output expected in RFCLI for this script is as follows:

```
---- 11:49:50.978 ------------------
DM_SM_REGISTER_REQ_T
        type =  2c02
        protocol_id =  01
        channel =  01
        outgoing_ok =  00
        security_level =  13
        psm =  00
```

3.5.5 Pair with Headset

The Pair with Headset routine in Figure 3.2 is covered by the procedure `PairWithHeadset`; this routine carries out the following:

- Stores the headset default PIN code of 1234 to be used in pairing
- Removes the headset device from the security manager
- Sets the security level

The TCL script is as follows:

```
# Pair with Headset
proc PairWithHeadset {} {
  global bd_addr.lap bd_addr.uap bd_addr.nap
  global pin_code pin_length
  after 3000

  # Store a default PIN to use when pairing
  set pin_code {0x31 0x32 0x33 0x34}
  set pin_length 4

  # Remove the device we're going to pair with from the security manager's
  # settings
  dm_sm_remove_device_req ${bd_addr.lap} ${bd_addr.uap} ${bd_addr.nap}

  # Set the security level
  set authentication_enabled 1
  set encryption_enabled 0
  dm_sm_set_sec_mode_req 3 0
}
```

The output expected in RFCLI for this script is as follows:

```
---- 11:49:53.993 ------------------
DM_SM_REMOVE_DEVICE_REQ_T
        type =  2c09
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02

---- 11:49:53.993 ------------------
DM_SM_REMOVE_DEVICE_CFM_T
        type =  2c15
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        success =  00

---- 11:49:53.993 ------------------
DM_SM_SET_SEC_MODE_REQ_T
        type =  2c07
        mode =  03
        mode3_enc =  00

---- 11:49:53.993 ------------------
DM_SM_SET_SEC_MODE_CFM_T
        type =  2c13
        phandle =  8000
        mode =  03
        mode3_enc =  00
        success =  01
```

3.5.6 Request to Open SDC Search

The Request to Open SDC Search routine in Figure 3.2 is covered by the procedure `StartSDPSearch`. This routine either requests to open a search on the headset or looks to see if a previous request to open has been confirmed. The selection variable `sel` passed by the calling routine determines whether an open search should be requested, or whether it should be confirmed. At the end of the routine the pairing status flag is set to one if pairing was successful. The TCL script for the routine is as follows:

```
# Request to Open SDC Search
proc StartSDPSearch {sel} {
  switch $sel {
    0 {SDC_OPEN_SEARCH_REQ - - - -}
    1 {set result [lindex [SDC_OPEN_SEARCH_CFM] 1]}
    2 {set result [lindex [sdc_open_search_req - - - -] 1]}
  }
  if {$sel > 0} {
    puts "Result SDC_OPEN_SEARCH_CFM: $result"
    # set the pair status depending on whether the pairing was a success
    if {$result == 0} {
    set pair_status 1
    } else {
      set pair_status 4
    }
  }
}
```

The output expected in RFCLI for this script when the routine is called to open the SDC search i.e. when variable `sel` is set to 0, is as follows:

```
---- 11:49:53.993 ------------------
SDC_OPEN_SEARCH_REQ_T
        type =  08
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
```

3.5.7 Wait for Link Key Request and Reject It

The Wait for Link Key Request and Reject It routine in Figure 3.2 is covered by the procedure `WaitLinkKeyRequestReject`. The script rejects the link key request because it initially has no link key to send. The TCL script for the routine is as follows:

```
# Wait for Link Key Request and Reject It
proc WaitLinkKeyRequestReject {} {
  # Wait for a link key request and reject it
  # It doesn't matter what we send down as a link key
  DM_SM_LINK_KEY_REQUEST_IND
  DM_SM_LINK_KEY_REQUEST_RES - - - {0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0} 0
}
```

The output expected in RFCLI for this script is as follows:

```
---- 11:49:55.275 ------------------
DM_SM_LINK_KEY_REQUEST_IND_T
        type =  2c16
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02

---- 11:49:55.275 ------------------
DM_SM_LINK_KEY_REQUEST_RES_T
        type =  2c0a
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        key [c] =  00
        valid =  00
```

3.5.8 Wait for PIN Code and Respond with Stored Value = 1234

The Wait for PIN Code and Respond with Stored Value = “1234”[1] routine in Figure 3.2 is covered by the procedure `WaitPinCodeRequest`. This routine waits for the PIN code request from the headset and then returns the default value of “1234”. If the headset has had a hard reset then the pin codes will match causing the devices to pair and authenticate. The TCL script for the routine is as follows:

```
# Wait for PIN Code and Respond with Stored Value = 1234
proc WaitPinCodeRequest {} {
  global pin_code pin_length
  DM_SM_PIN_REQUEST_IND
  DM_SM_PIN_REQUEST_RES - - - $pin_length $pin_code
}
```

The output expected in RFCLI for this script is as follows:

```
---- 11:49:55.290 ------------------
DM_SM_PIN_REQUEST_IND_T
        type =  2c17
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02

---- 11:49:55.290 ------------------
DM_SM_PIN_REQUEST_RES_T
        type =  2c0b
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        pin_length =  04
        pin [c] =  31
        pin [c] =  32
        pin [c] =  33
        pin [c] =  34
        pin [c] =  00
        pin [c] =  00
        pin [c] =  00
        pin [c] =  00
        pin [c] =  00
        pin [c] =  00
        pin [c] =  00
        pin [c] =  00
        pin [c] =  00
        pin [c] =  00
        pin [c] =  00
        pin [c] =  00
```

Note:

[1] The pin number “1234” is ASCII coded

3.5.9 Wait for Link Key

The Wait for Link Key routine in Figure 3.2 is covered by the procedure `WaitLinkKeyRequest`. This routine stores the headset device and the new link key received from the security manager. The TCL script is as follows:

```
# Wait for Link Key
proc WaitLinkKeyRequest {} {
  global key
  DM_SM_LINK_KEY_IND

  # Store the link key in the security manager
  dm_sm_add_device_req - - - 1 1 1 $key
}
```

The output expected in RFCLI for this script is as follows:

```
---- 11:49:55.587 ------------------
DM_SM_LINK_KEY_IND_T
        type =  2c18
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        key_type =  00
        key [c] =  91
        key [c] =  02
        key [c] =  09
        key [c] =  5b
        key [c] =  b5
        key [c] =  ff
        key [c] =  e0
        key [c] =  73
        key [c] =  04
        key [c] =  da
        key [c] =  5b
        key [c] =  0b
        key [c] =  09
        key [c] =  f2
        key [c] =  97
        key [c] =  d1

---- 11:49:55.587 ------------------
DM_SM_ADD_DEVICE_REQ_T
        type =  2c08
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        update_trust_level =  01
        trusted =  01
        update_link_key =  01
        link_key [c] =  91
        link_key [c] =  02
        link_key [c] =  09
        link_key [c] =  5b
        link_key [c] =  b5
        link_key [c] =  ff
        link_key [c] =  e0
        link_key [c] =  73
        link_key [c] =  04
        link_key [c] =  da
        link_key [c] =  5b
        link_key [c] =  0b
        link_key [c] =  09
        link_key [c] =  f2
        link_key [c] =  97
        link_key [c] =  d1
```

```
---- 11:49:55.603 ------------------
DM_SM_ADD_DEVICE_CFM_T
        type =  2c14
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        success =  01
```

3.5.10 Wait for Open SDC Search Confirmed

The following line of TCL script that appears in the procedure `mainfunction` covers the `Wait for Open SDC Search Confirmed` routine in Figure 3.2:

```
# Wait for Open SDC Search Confirmed
StartSDPSearch 1
```

The `Wait for Open SDC Search Confirmed` routine is essentially a call routine to the `StartSDPSearch` procedure documented in Section 3.5.6 but a different value for the `sel` variable is used this time it is set to one. The TCL script is listed earlier in Section 3.5.6 and as part of the opening of the SDC search being confirmed an ACL connection will be automatically established with the headset and its remote feature set supplied. The output expected in RFCLI for this script is as follows:

```
---- 11:49:55.618 ------------------
DM_ACL_OPENED_IND_T
        type =  280d
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        incoming =  00
        dev_class =  00

---- 11:49:55.900 ------------------
DM_HCI_READ_REMOTE_FEATURES_COMPLETE_T
        type =  42a
        phandle =  8000
        status =  00
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        features [c] =  ffff
        features [c] =  0f
        features [c] =  00
        features [c] =  00

---- 11:49:56.165 ------------------
SDC_OPEN_SEARCH_CFM_T
        type =  09
        phandle =  8000
        result =  00
Result SDC_OPEN_SEARCH_CFM: 0x0
```

3.5.11 Search for HID Service Record on the Headset

The Search for HID Service Record on the Headset routine in Figure 3.2 is covered by the procedure `SearchHIDServiceRecord`. This routine is an example of the use of the SDP_ServiceSearch transaction outlined in the Bluetooth Specification v1.1 and is intended to show a failure, it causes a search for the HID service record on the headset. The headset does not contain the HID service record and therefore should return zero service records. The TCL script is as follows:

```
# Search for HID Service Record on the Headset
proc  SearchHIDServiceRecord {} {
  global UsingPrimitives

  #search for HID service records because this should
  #give a failure as we are communicating with Headset
  if {$UsingPrimitives} {
    SDC_SERVICE_SEARCH_REQ - - - - 5 {0x35 0x3 0x19 0x11 0x24} 1
    set result [SDC_SERVICE_SEARCH_CFM]
  } else {
  set result [sdc_service_search_req - - - - 5 {0x35 0x3 0x19 0x11 0x24} 1]
  }
  puts "Service Record Search Result HID: $result"
}
```

The output expected in RFCLI for this script is as follows:

```
---- 11:49:56.165 ------------------
SDC_SERVICE_SEARCH_REQ_T
        type =  01
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        size_srch_pttrn =  05
        max_num_recs =  01
 35 03 19 11 24

---- 11:49:56.212 ------------------
SDC_SERVICE_SEARCH_CFM_T
        type =  02
        phandle =  8000
        num_recs_ret =  00
        size_rec_list =  00
        response =  11
        err_code =  00
        size_err_info =  00
Service Record Search Result HID: 0x8000 0x0 0x0 0x0 0x11 0x0 0x0 0x0
```

Note:

> The `response = 11` means that there is no response data i.e. the service being search for does not exist on the server. For the meaning of this response and other error code refer to file **sdc_prim.h**.

3.5.12 Search for Generic Audio Service Record on the Headset

The Search for Generic Audio Service Record on the Headset routine in Figure 3.2 is covered by the procedure `SearchGenericAudioServiceRecord`. This routine is an example of the use of the SDP_ServiceSearch transaction outlined in the Bluetooth Specification v1.1 and searches for the generic audio service record on the headset SDP server and is therefore successful. The TCL script is as follows:

```
# Search for Generic Audio Service on the Headset
proc SearchGenericAudioServiceRecord {} {
  global UsingPrimitives

  #search for Generic Audio service records
  if {$UsingPrimitives} {
    SDC_SERVICE_SEARCH_REQ - - - - 5 {0x35 0x3 0x19 0x12 0x03} 1
    set result [SDC_SERVICE_SEARCH_CFM]
  } else {
    set result [sdc_service_search_req - - - - 5 {0x35 0x3 0x19 0x12 0x03}
1]
  }
  puts "Service Record Search Result GA: $result"
}
```

The output expected in RFCLI for this script is as follows:

```
---- 11:49:56.212 ------------------
SDC_SERVICE_SEARCH_REQ_T
        type =  01
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        size_srch_pttrn =  05
        max_num_recs =  01
 35 03 19 12 03

---- 11:49:56.275 ------------------
SDC_SERVICE_SEARCH_CFM_T
        type =  02
        phandle =  8000
        num_recs_ret =  01
        size_rec_list =  04
        response =  00
        err_code =  00
        size_err_info =  00
 00 01 00 00
Service Record Search Result GA: 0x8000 0x1 0x4 {0x00 0x01 0x00 0x00} 0x0
0x0 0x0 0x0
```

3.5.13 Search for Attributes on the Headset

The Search for Attributes on the Headset routine in Figure 3.2 is covered by the procedure `ServiceAttributes`. This routine is an example of the use of the SDP_ServiceAttribute transaction outlined in the Bluetooth Specification v1.1 and at first gets the service record handle for the headset by searching for it on the headset using the `SearchHeadsetServiceRecord` procedure listed below:

```
proc  SearchHeadsetServiceRecord {} {
  global UsingPrimitives

  #search for Headset service records
  if {$UsingPrimitives} {
    SDC_SERVICE_SEARCH_REQ - - - - 5 {0x35 0x3 0x19 0x11 0x08} 1
    set result [SDC_SERVICE_SEARCH_CFM]
  } else {
    set result [sdc_service_search_req - - - - 5 {0x35 0x3 0x19 0x11 0x08}
1]
  }
  puts "Service Record Search Result HS: $result"
  #Strip off the service record handle
  set srh [lindex $result 3]
  set retval0 [expr {0x01000000 * [lindex $srh 0]}]
  set retval1 [expr {0x010000 * [lindex $srh 1]}]
  set retval2 [expr {0x0100 * [lindex $srh 2]}]
  set retval3 [lindex $srh 3]
  set retval [expr {$retval0 + $retval1 + $retval2 + $retval3}]
  return $retval
}
```

Once the service record handle for the headset is obtained it uses this handle to search and confirm that the attribute IDs `ServiceClassIDList`, `ProtocolDescriptorList`, `BluetoothProfileDescriptorList`, `ServiceName` and `Remote Audio Volume Control` are present. These attribute searches should all be successful as they present in the headset service record supplied with BlueLab v2.5, the headset service record is coded in BlueLab v2.5 as follows:

```
/*
    This structure defines the service record for the Headset; it contains
    a blank space for the RFCOMM server channel since this will be filled
    in at run time
    Note that attrIds are specified as being 16-bit ints
*/
static const uint8 serviceRecord[] =
{

  /* Service class ID list */
  0x09,0x00,0x01, /* AttrID , ServiceClassIDList */
  0x35,0x06, /* 6 bytes in total DataElSeq */
  0x19,0x11,0x08,/* 2 byte UUID, Service class = headset */
  0x19,0x12,0x03,/* 2 byte UUID Service class = GenericAudio */

  /* protocol descriptor list */
  0x09,0x00,0x04,/* AttrId ProtocolDescriptorList */
  0x35,0x0c, /* 11 bytes in total DataElSeq */
  0x35,0x03, /*3 bytes in DataElSeq */
  0x19, 0x01,0x00,/* 2 byte UUID, Protocol = L2CAP */

  0x35,0x05, /* 4 bytes in DataElSeq */
  0x19, 0x00,0x03,  /* 1 byte UUID Protocol = RFCOMM */
  0x08, 0x00, /* 1 byte UINT - server channel template value 0 - to be
               filled in by app */

  /* profile descriptor list */
  0x09,0x00,0x09, /* AttrId, ProfileDescriptorList */
  0x35,0x08, /* 10 bytes in total DataElSeq */
  0x35,0x06, /* 6 bytes in total DataElSeq */
  0x19, 0x11,0x08, /* 2 byte UUID, Service class = Headset */
  0x09, 0x01,0x00, /* 2 byte uint, version = 100 */
  /* service name */
  0x09, 0x01, 0x00, /* AttrId - Service Name */
  0x25, 0x07, /* 7 byte string */
  'H','e','a','d','s','e','t',

  /* remote audio volume control */
  0x09, 0x03, 0x02, /* AttrId - remote audio volume control */
  0x28, 0x01 /* boolean - TRUE we do support remote audio volume control */
};
```

The actual TCL for the Search for Attributes on the Headset routine is as follows:

```
# Search for Attributes on the Headset
proc ServiceAttributes {} {
  global UsingPrimitives
  global bd_addr.lap bd_addr.uap bd_addr.nap

  #Get Service Record handle for the Headset
  set srh [SearchHeadsetServiceRecord]
  puts [format "Service Record Handle: 0x%x" $srh]

  #Search for the ServiceClassIDList Attribute (Attribute ID = 0x0001) on
  #Headset
  if {$UsingPrimitives} {
    SDC_SERVICE_ATTRIBUTE_REQ - ${bd_addr.lap} ${bd_addr.uap}
${bd_addr.nap} $srh 5 {0x35 0x3 0x09 0x00 0x01} 10
    set result [SDC_SERVICE_ATTRIBUTE_CFM]
  } else {
    set result [sdc_service_attribute_req - ${bd_addr.lap} ${bd_addr.uap}
${bd_addr.nap} $srh 5 {0x35 0x3 0x09 0x00 0x01} 10]
  }
  puts "Service Attribute Result for ServiceClassIDList: $result"

  #Search for the ProtocolDescriptorList Attribute (Attribute ID = 0x0004)
  #on Headset
  if {$UsingPrimitives} {
    SDC_SERVICE_ATTRIBUTE_REQ - ${bd_addr.lap} ${bd_addr.uap}
${bd_addr.nap} $srh 5 {0x35 0x3 0x09 0x00 0x04} 10
    set result [SDC_SERVICE_ATTRIBUTE_CFM]
  } else {
    set result [sdc_service_attribute_req - ${bd_addr.lap} ${bd_addr.uap}
${bd_addr.nap} $srh 5 {0x35 0x3 0x09 0x00 0x04} 10]
  }

  puts "Service Attribute Result for ProtocolDescriptorList: $result"

  #Search for the BluetoothProfileDescriptorList Attribute (Attribute ID =
  #0x0009) on Headset
  if {$UsingPrimitives} {
    SDC_SERVICE_ATTRIBUTE_REQ - ${bd_addr.lap} ${bd_addr.uap}
${bd_addr.nap} $srh 5 {0x35 0x3 0x09 0x00 0x09} 10
    set result [SDC_SERVICE_ATTRIBUTE_CFM]
  } else {
    set result [sdc_service_attribute_req - ${bd_addr.lap} ${bd_addr.uap}
${bd_addr.nap} $srh 5 {0x35 0x3 0x09 0x00 0x09} 10]
  }

  puts "Service Attribute Result for BluetoothProfileDescriptorList:
$result"

  #Search for the ServiceName Attribute (Attribute ID = 0x0100) on Headset
  if {$UsingPrimitives} {
    SDC_SERVICE_ATTRIBUTE_REQ - ${bd_addr.lap} ${bd_addr.uap}
${bd_addr.nap} $srh 5 {0x35 0x3 0x09 0x01 0x00} 10
    set result [SDC_SERVICE_ATTRIBUTE_CFM]
  } else {
    set result [sdc_service_attribute_req - ${bd_addr.lap} ${bd_addr.uap}
${bd_addr.nap} $srh 5 {0x35 0x3 0x09 0x01 0x00} 10]
  }

  puts "Service Attribute Result for ServiceName: $result"

  #Search for the Remote Audio Volume Control Attribute (Attribute ID =
  #0x0302) on Headset
  if {$UsingPrimitives} {
    SDC_SERVICE_ATTRIBUTE_REQ - ${bd_addr.lap} ${bd_addr.uap}
${bd_addr.nap} $srh 5 {0x35 0x3 0x09 0x03 0x02} 10
    set result [SDC_SERVICE_ATTRIBUTE_CFM]
  } else {
    set result [sdc_service_attribute_req - ${bd_addr.lap} ${bd_addr.uap}
${bd_addr.nap} $srh 5 {0x35 0x3 0x09 0x03 0x02} 10]
  }

  puts "Service Attribute Result for Remote Audio Volume Control: $result"
}
```

The overall output expected in RFCLI for this script which includes the `SearchHeadsetServiceRecord` procedure is as follows:

```
---- 11:49:56.275 ------------------
SDC_SERVICE_SEARCH_REQ_T
        type =  01
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        size_srch_pttrn =  05
        max_num_recs =  01
 35 03 19 11 08

---- 11:49:56.321 ------------------
SDC_SERVICE_SEARCH_CFM_T
        type =  02
        phandle =  8000
        num_recs_ret =  01
        size_rec_list =  04
        response =  00
        err_code =  00
        size_err_info =  00
 00 01 00 00

Service Record Search Result HS: 0x8000 0x1 0x4 {0x00 0x01 0x00 0x00} 0x0
0x0 0x0 0x0
Service Record Handle: 0x10000

---- 11:49:56.321 ------------------
SDC_SERVICE_ATTRIBUTE_REQ_T
        type =  03
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        svc_rec_hndl =  10000
        size_attr_list =  05
        max_num_attr =  0a
 35 03 09 00 01
```

```
---- 11:49:56.400 ------------------
SDC_SERVICE_ATTRIBUTE_CFM_T
        type =  04
        phandle =  8000
        size_attr_list =  0b
        response =  00
        err_code =  00
        size_err_info =  00
 09 00 01 35 06 19 11 08 19 12 03

Service Attribute Result for ServiceClassIDList: 0x8000 0xb {0x09 0x00 0x01
0x35 0x06 0x19 0x11 0x08 0x19 0x12 0x03} 0x0 0x0 0x0 0x0

---- 11:49:56.400 ------------------
SDC_SERVICE_ATTRIBUTE_REQ_T
        type =  03
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        svc_rec_hndl =  10000
        size_attr_list =  05
        max_num_attr =  0a
 35 03 09 00 04

---- 11:49:56.493 ------------------
SDC_SERVICE_ATTRIBUTE_CFM_T
        type =  04
        phandle =  8000
        size_attr_list =  11
        response =  00
        err_code =  00
        size_err_info =  00
 09 00 04 35 0c 35 03 19 01 00 35 05 19 00 03 08 01

Service Attribute Result for ProtocolDescriptorList: 0x8000 0x11 {0x09 0x00
0x04 0x35 0x0c 0x35 0x03 0x19 0x01 0x00 0x35 0x05 0x19 0x00 0x03 0x08 0x01}
0x0 0x0 0x0 0x0

---- 11:49:56.493 ------------------
SDC_SERVICE_ATTRIBUTE_REQ_T
        type =  03
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        svc_rec_hndl =  10000
        size_attr_list =  05
        max_num_attr =  0a
 35 03 09 00 09

---- 11:49:56.571 ------------------
SDC_SERVICE_ATTRIBUTE_CFM_T
        type =  04
        phandle =  8000
        size_attr_list =  0d
        response =  00
        err_code =  00
        size_err_info =  00
 09 00 09 35 08 35 06 19 11 08 09 01 00

Service Attribute Result for BluetoothProfileDescriptorList: 0x8000 0xd
{0x09 0x00 0x09 0x35 0x08 0x35 0x06 0x19 0x11 0x08 0x09 0x01 0x00} 0x0 0x0
0x0 0x0
```

```
---- 11:49:56.571 ------------------
SDC_SERVICE_ATTRIBUTE_REQ_T
        type =  03
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        svc_rec_hndl =  10000
        size_attr_list =  05
        max_num_attr =  0a
 35 03 09 01 00

---- 11:49:56.665 ------------------
SDC_SERVICE_ATTRIBUTE_CFM_T
        type =  04
        phandle =  8000
        size_attr_list =  0c
        response =  00
        err_code =  00
        size_err_info =  00
 09 01 00 25 07 48 65 61 64 73 65 74

Service Attribute Result for ServiceName: 0x8000 0xc {0x09 0x01 0x00 0x25
0x07 0x48 0x65 0x61 0x64 0x73 0x65 0x74} 0x0 0x0 0x0 0x0

---- 11:49:56.665 ------------------
SDC_SERVICE_ATTRIBUTE_REQ_T
        type =  03
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        svc_rec_hndl =  10000
        size_attr_list =  05
        max_num_attr =  0a
 35 03 09 03 02

---- 11:49:56.712 ------------------
SDC_SERVICE_ATTRIBUTE_CFM_T
        type =  04
        phandle =  8000
        size_attr_list =  05
        response =  00
        err_code =  00
        size_err_info =  00
 09 03 02 28 01
Service Attribute Result for Remote Audio Volume Control: 0x8000 0x5 {0x09
0x03 0x02 0x28 0x01} 0x0 0x0 0x0 0x0
```

3.5.14 Search for Service Attributes on the Headset

The Search for Service Attributes on the Headset routine in Figure 3.2 is covered by the procedure `ServiceSearchAttributes`. This routine is an example of the use of the SDP_ServiceSearchAttribute transaction outlined in the Bluetooth Specification v1.1 and is seen as a combination of carrying out an SDP_ServiceSearch transaction followed by an SDP_ServiceAttribute transaction. The routine first needs to get the service record handle for the headset by searching for it on the headset using the `SearchHeadsetServiceRecord`, which is documented in Section 3.5.13. The TCL script is as follows:

```
# Search for Service Attributes on the Headset
proc ServiceSearchAttributes {} {
  global UsingPrimitives
  global bd_addr.lap bd_addr.uap bd_addr.nap

  #Get Service Record handle for the Headset
  set srh [SearchHeadsetServiceRecord]
  puts [format "Service Record Handle: 0x%x" $srh]

  #search for Headset service (UUID = 0x1108 record and ServiceName
  #Attribute (Attribute ID = 0x0100) on Headset
  if {$UsingPrimitives} {
    SDC_SERVICE_SEARCH_ATTRIBUTE_REQ - ${bd_addr.lap} ${bd_addr.uap}
${bd_addr.nap} 5 {0x35 0x3 0x19 0x11 0x08} 5 {0x35 0x3 0x09 0x01 0x00} 10
    set result [SDC_SERVICE_SEARCH_ATTRIBUTE_CFM]
  } else {
    set result [sdc_service_search_attribute_req - ${bd_addr.lap}
${bd_addr.uap} ${bd_addr.nap} 5 {0x35 0x3 0x19 0x11 0x08} 5 {0x35 0x3 0x09
0x01 0x00} 10]
  }

  puts "Service Search Attribute Result for Headset and ServiceName:
$result"
}
```

The overall output expected in RFCLI for this script, which includes the `SearchHeadsetServiceRecord` procedure is as follows:

```
Service Record Search Result HS: 0x8000 0x1 0x4 {0x00 0x01 0x00 0x00} 0x0
0x0 0x0 0x0
Service Record Handle: 0x10000

---- 11:49:56.759 ------------------
SDC_SERVICE_SEARCH_ATTRIBUTE_REQ_T
        type =  05
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        size_srch_pttrn =  05
        size_attr_list =  05
        max_num_attr =  0a
 35 03 19 11 08
 35 03 09 01 00

---- 11:49:56.853 ------------------
SDC_SERVICE_SEARCH_ATTRIBUTE_CFM_T
        type =  06
        phandle =  8000
        total_response_size =  0f
        size_attr_list =  0c
        more_to_come =  00
        response =  00
        err_code =  00
        size_err_info =  00
 09 01 00 25 07 48 65 61 64 73 65 74

Service Search Attribute Result for Headset and ServiceName: 0x8000 0xf 0xc
{0x09 0x01 0x00 0x25 0x07 0x48 0x65 0x61 0x64 0x73 0x65 0x74} 0x0 0x0 0x0
0x0 0x0
```

3.5.15 Show Range of Attributes Available on the Headset

The Show Range of Attributes Available on the Headset routine in Figure 3.2 is covered by a procedure call to the library function `sdc_range_search`. It carries out an SDP_ServiceSearch transaction and in this example it looks for the headset service (0x1108) and then follows up with an SDP_ServiceAttribute transaction but for a range of attribute IDs set at 0x0000 to 0xffff. Essentially it lists the entire attribute IDs on the headset and the results shown in **RFCLI_OutputPrimitives.txt** and **RFCLI_OutputLibrary.tcl** can be compared to the headset service record coded in C shown in Section 3.5.13. The line of TCL script in the mainroutine procedure that is responsible for the Show Range of Attributes Available on the Headset routine is as follows:

```
# Show Range of Attributes Available on the Headset
sdc_range_search 0x1108
```

The output expected in RFCLI for this script:

```
---- 11:49:56.853 ------------------
SDC_SERVICE_SEARCH_ATTRIBUTE_REQ_T
        type =  05
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        size_srch_pttrn =  05
        size_attr_list =  07
        max_num_attr =  3e8
 35 03 19 11 08
 35 05 0a 00 00 ff ff
---- 11:49:56.931 ------------------
SDC_SERVICE_SEARCH_ATTRIBUTE_CFM_T
        type =  06
        phandle =  8000
        total_response_size =  45
        size_attr_list =  42
        more_to_come =  00
        response =  00
        err_code =  00
        size_err_info =  00
09 00 00 0a 00 01 00 00 09 00 01 35 06 19 11 08 19 12 03 09 00 04 35 0c 35
03 19 01 00 35 05 19 00 03 08 01 09 00 09 35 08 35 06 19 11 08 09 01 00 09
01 00 25 07 48 65 61 64 73 65 74 09 03 02 28 01
```

3.5.16 Terminate a Search

The Terminate a Search routine in Figure 3.2 is covered by the procedure `TerminateSearch`. The purpose of this routine is set up a service search and to cancel it. This is done to demonstrate the functionality of the SDC_TERMINATE_PRIMITIVE_REQ primitive. The TCL script is as follows:

```
# Terminate a Search
proc TerminateSearch {} {
  global bd_addr.lap bd_addr.uap bd_addr.nap

  #Get Service Record handle for the Headset
  set srh [SearchHeadsetServiceRecord]
  puts [format "Service Record Handle: 0x%x" $srh]

  #search for Headset service (UUID = 0x1108 record and ServiceName
  #Attribute (Attribute ID = 0x0100) on Headset then terminate the search
  SDC_SERVICE_SEARCH_ATTRIBUTE_REQ - ${bd_addr.lap} ${bd_addr.uap}
${bd_addr.nap} 5 {0x35 0x3 0x19 0x11 0x08} 5 {0x35 0x3 0x09 0x01 0x00} 10
  SDC_TERMINATE_PRIMITIVE_REQ
}
```

The output expected in RFCLI for this script:

```
Service Record Search Result HS: 0x8000 0x1 0x4 {0x00 0x01 0x00 0x00} 0x0
0x0 0x0 0x0
Service Record Handle: 0x10000

---- 11:49:56.978 ------------------
SDC_SERVICE_SEARCH_ATTRIBUTE_REQ_T
        type =  05
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
        size_srch_pttrn =  05
        size_attr_list =  05
        max_num_attr =  0a
 35 03 19 11 08
 35 03 09 01 00

---- 11:49:56.978 ------------------
SDC_TERMINATE_PRIMITIVE_REQ_T
        type =  07
        phandle =  8000
```

3.5.17 Unregister AG Service Record

The Unregister AG Service Record routine in Figure 3.2 is covered by the procedure `UnRegisterAGServiceRecord`. This routine demonstrates how to unregister the service that has been previously registered within the SDP server database by the Register AG Service Record routine outlined in Section 3.5.3. The calling routine must supply the service record handle of the service it would like removing from the database, in this case it is the service record handle variable returned by the Register AG Service Record routine and is stored in the variable `agrecordhandle` that is local to the `mainroutine` procedure. The TCL script is as follows:

```
# Unregister AG Service Record
proc UnRegisterAGServiceRecord {ph recordhandle} {
  global UsingPrimitives
  if {$UsingPrimitives} {
    SDS_UNREGISTER_REQ $ph $recordhandle
    SDS_UNREGISTER_CFM
  } else {
    sds_unregister_req $ph $recordhandle
  }
}
```

The output expected in RFCLI for this script:

```
---- 11:49:56.978 ------------------
SDS_UNREGISTER_REQ_T
        type =  10
        phandle =  8000
        svc_rec_hndl =  10000

---- 11:49:56.993 ------------------
SDS_UNREGISTER_CFM_T
        type =  11
        phandle =  8000
        svc_rec_hndl =  10000
        result =  00
```

3.5.18 Close SDC Search

The `Close SDC Search` routine in Figure 3.2 is covered by the following lines of TCL script that is present in the `mainfunction` procedure:

```
# Close SDC Search
  if {$UsingPrimitives} {
    SDC_CLOSE_SEARCH_REQ
    set result [lindex [SDC_CLOSE_SEARCH_IND] 1]
  } else {
    set result [lindex [sdc_close_search_req] 1]
  }
  switch $result {
    0x0 {puts "Connection disconnected by SDS Server"}
    0x1 {puts "Connection disconnected by SDS Client"}
  }
```

The routine closes down the client search on the SDP database and removes the ACL connection, with the switch statement indicating whether the client or the server closed the link. The output expected in RFCLI for this script:

```
---- 11:49:56.993 ------------------
SDC_CLOSE_SEARCH_REQ_T
        type =  0a
        phandle =  8000

---- 11:49:57.040 ------------------
SDC_CLOSE_SEARCH_IND_T
        type =  0c
        phandle =  8000
        result =  01
Connection disconnected by SDS Client
RFCLI>
---- 11:49:57.056 ------------------
DM_ACL_CLOSED_IND_T
        type =  280e
        phandle =  8000
        BD_ADDR_T
                lap =  10e47
                uap =  5b
                nap =  02
```

4 Document References

Document:	Reference, Date:
Bluetooth Specification - Core	V1.1, v1.1, 22 February 2001
Bluetooth Specification – Profiles	V1.1, v1.1, 22 February 2001
BlueStack User Manual	C6066-UM-001, v1.6
RFCLI User Manual	bcore-ug-003Pa, a, September 2002
Bluetooth Connect Without Cables – Jennifer Bray and Charles F Sturman	ISBN 0-13-089840-6, Prentice Hall PTR, 2001
Tcl and the Tk Toolkit – John K Ousterhout	ISBN 0-201-63337-X, Addison-Wesley, 1994
Accessing RFCOMM Using RFCLI and TCL	bcore-an-006Pa, a, September 2002
Bluetooth Assigned Number – Service Discovery Protocol	See Website: http://www.bluetooth.org/assigned-numbers/sdp.htm
Bluetooth Security White Paper – Bluetooth SIG Security Expert Group	v1.00, 19 April 2002

Acronyms and Definitions

ACL	Asynchronous ConnectionLess
API	Application Programming Interface
BlueCore™	Group term for CSR's range of Bluetooth chips
Bluetooth™	Set of technologies providing audio and data transfer over short-range radio connections
L2CAP	Logical Link Control and Adaptation Protocol
MTU	Maximum Transmission Unit
PDU	Protocol Data Unit
RFCLI	RFCOMM Command Line Interface
RFCOMM	Protocol for RS-232 serial cable emulation
SDC	Service Discovery Client
SDD	Service Discovery Database
SDP	Service Discovery Protocol
SDS	Service Discovery Server
TCL	Tool Command Language
UUID	Universally Unique Identifier

Record of Changes

Date:	Revision	Reason for Change:
11 DEC 02	a	Original publication of this document. (CSR reference: bcore-an-007Pa)

Accessing Service Discovery Using RFCLI and TCL

Application Note

bcore-an-007Pa

December 2002

CSR's products are not authorised for use in life-support or safety-critical applications.