
S60 Platform: Bluetooth API Developer's Guide

Version 2.1
June 27, 2008

S60 platform

Legal notice

Copyright © 2004–2008 Nokia Corporation. All rights reserved.

Nokia and Forum Nokia are registered trademarks of Nokia Corporation. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this document at any time, without notice.

License

A license is hereby granted to download and print a copy of this document for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

1.	Introduction	7
1.1	Development tools and code examples	7
1.2	Purpose of this document	8
2.	Overview of Bluetooth architecture in the S60 platform	9
2.1	Bluetooth API blocks	10
2.1.1	Bluetooth sockets	11
2.1.2	Bluetooth Service Discovery Database	12
2.1.3	Bluetooth Service Discovery Agent	12
2.1.4	Bluetooth Security Manager	12
2.1.5	Bluetooth UI	13
2.2	Common Bluetooth data types	13
2.2.1	Bluetooth device address	13
2.2.2	Universally Unique Identifier (UUID)	13
2.2.3	Service record	14
2.2.4	Service class and profiles	14
2.2.5	Service record handle	14
2.2.6	Service attribute ID	14
2.2.7	SDP database	14
2.2.8	Data elements	14
2.2.9	CSdpAttrValue data type values	15
2.2.10	Service search patterns	15
2.2.11	Attribute search patterns	16
2.2.12	Attribute ranges	16
2.3	Bluetooth API v2 architecture	16
2.4	Platform security — effects on Bluetooth application development	16
3.	Device discovery	19
3.1	Using the Bluetooth UI	19
3.1.1	Limitations of the device selection notifier plug-in	21
3.2	Running a device discovery with RHostResolver	21
3.2.1	Getting the address of a remote device	22
3.2.2	Getting the name of a remote device	22
3.2.3	Setting the Inquiry Access Code (IAC)	22
3.2.4	Device discovery in the Bluetooth point-to-multipoint example	23
3.2.5	Notes about the RHostResolver discovery	25
4.	Service discovery	27
4.1	CSdpAgent and MSdpAgentNotifier classes	27
4.1.1	Creating the instance of CSdpAgent	28

4.1.2	Running the service search	28
4.1.3	Running the attribute search	28
4.1.4	Bluetooth Chat example	29
5.	Discoverability and advertising	32
5.1	Advertising a service	32
5.1.1	The SDP database	32
5.1.2	Creating a service record	32
5.1.3	Populating the service record	33
5.1.4	CSdpAttrValueDES and CSdpAttrValueDES	33
5.1.5	Removing the service advertisement	36
5.1.6	Shutting down the SDP database	37
5.2	Changing the discoverability mode	37
5.3	Bluetooth registry	37
5.3.1	Adding a device	38
5.3.2	Creating a view	38
5.3.3	Retrieving results	39
5.3.4	Deleting and unpairing devices	40
5.3.5	Modifying devices	40
5.3.6	Communication port settings	41
5.3.7	Overview of the older architecture	41
6.	Communication using sockets	42
6.1	CBluetoothSocket	42
6.2	Device name length	43
6.3	Master, slave, and role switching	43
6.4	Options and protocol strings	44
6.5	Listening sockets	44
6.5.1	Listening sockets in Bluetooth API v1 architecture	44
6.6	Transmitting socket	46
6.7	Sending data	46
6.8	Receiving data	46
6.9	Shutting down the service	47
7.	Communication using OBEX	48
7.1	Defining Bluetooth protocol	48
7.2	OBEX objects	49
7.3	OBEX server notification mechanism	50
7.4	OBEX client	51
7.5	OBEX server	52
7.6	Easy way of using OBEX with Send UI	52
7.6.1	Send UI API usage example	53

8.	Security and configurations	57
8.1	Bluetooth Security Manager.....	57
8.1.1	Service security in Bluetooth API v2 architecture.....	57
8.1.2	Service security in Bluetooth API v1 architecture.....	58
8.2	Bluetooth Publish and Subscribe	59
8.2.1	Bluetooth categories and property keys	59
8.2.2	Defining keys in the Bluetooth Control category	61
8.2.3	Getting the values.....	61
8.3	Checking if Bluetooth is supported	61
8.4	Checking Bluetooth power mode	62
8.4.1	Asking the user to turn Bluetooth power on	62
9.	Terms and abbreviations	64
10.	References	65
Appendix A.	Bluetooth API v2 changes	66
Appendix B.	Bluetooth API changes in S60 3rd Edition	68
	Evaluate this resource	71

Change history

August 27, 2004	Version 1.0	Initial document release
April 7, 2005	Version 1.1	<p>Section 10.2 updated to cover Series 80 devices and the Nokia 7710 multimedia smartphone</p> <p>References to the Series 90 Developer Platform 2.0 replaced with references to the Nokia 7710 multimedia smartphone.</p>
December 22, 2006	Version 2.0	<p>Document updated to reflect the changes and additions in S60 3rd Edition.</p> <p>Document name also changed. The previous name was "Symbian OS: Designing Bluetooth Applications In C++".</p>
June 27, 2008	Version 2.1	Updates and corrections in Sections 3.1 and 3.2.5.

1. Introduction

Third-party application developers can create their own Bluetooth applications using many available public APIs. Applications range from simple chat applications to demanding multiplayer games. Multiple connections from one device to other devices are also possible, enabling point-to-multipoint (PMP) applications.

Additionally, Bluetooth offers possibilities for the use of wireless accessory appliances. A developer can write smartphone applications that use Bluetooth for a variety of wireless accessories such as bar-code readers, digital pens, health-monitoring devices, and Global Positioning System (GPS) receivers. (Developing Bluetooth applications to systems other than those used by S60 devices is beyond the scope of this document.)

Beginning with Symbian OS v8.0a (S60 2nd Edition, Feature Pack 2), the Bluetooth APIs have experienced major architectural improvements (referred to as Bluetooth API v2 in this document). Headset and Basic Imaging Profile are supported in the new architecture. This document is written to describe both architectures. See Table 6 in Appendix A for a list of changes in the Bluetooth architecture.

Most of the changes in Symbian OS v9.1 will not affect Bluetooth developers. However, the Symbian platform security that has been introduced does have some consequences, not necessarily to the Bluetooth APIs but to the way they can be used. Bluetooth 1.2 and Remote SIM Access Profile (which is needed to access subscriber identity module [SIM] information from Bluetooth devices, such as car kits) are supported in S60 3rd Edition.

This document aims to provide a holistic view of the available Bluetooth APIs (both Symbian APIs and S60 APIs) in the S60 platform, as well as to describe the changes between different S60 releases (including the new and old Bluetooth architectures). Because most of the Bluetooth APIs are delivered by Symbian, sections of this document can also be applied to other Symbian OS devices.

1.1 Development tools and code examples

To develop Bluetooth applications in Symbian C++, developers will need to download and install an SDK compatible with the platform they are developing for. For SDKs, please visit <http://www.forum.nokia.com/tools>, where C++ SDKs can be downloaded free of charge. The SDKs include all the key functionality required for application development (documentation, API reference, add-on tools, device emulator, and target compiler) except for an integrated development environment (IDE) that must also be acquired in order to write the code. For information on how to set up the Bluetooth development environment and run Bluetooth applications on the device emulator, refer to the SDK Help.

This document refers to a number of Bluetooth-related examples, such as the Chat example from S60 3rd Edition, and Bluetooth OBEX and Bluetooth PMP examples. There are a number of Bluetooth examples available in the S60 SDKs. For communication over serial port (RFCOMM), the S60 3rd Edition SDK provides a Chat example (the example provides both Bluetooth and TCP/IP interfaces). In S60 1st and 2nd Edition SDKs, there is a Bluetooth point-to-point example (for RFCOMM) and OBEX, HCI, and Discovery/Advertiser examples.

[S60 Platform: Bluetooth Point-to-Multipoint Example](#) [5] (also discussed as the Bluetooth PMP example) is available at www.forum.nokia.com. It demonstrates the communication between multiple devices in a piconet (one master and multiple slaves).

[S60 Platform: Bluetooth OBEX Example](#) [4] demonstrates the Symbian OBEX APIs.

1.2 Purpose of this document

This document provides developers who already have existing knowledge of Bluetooth technology with enough information to develop Bluetooth applications in C++ for the S60 platform. Developing Bluetooth applications in Java™ programming language is beyond the scope of this document.

The chapters that follow will describe how the Symbian OS and S60 APIs make Bluetooth functionality available for application developers, not how the Bluetooth protocols work. For more information about Bluetooth protocols, see the *Bluetooth Core Specification* at www.bluetooth.org [1] (registration required).

The terms "client" and "server" are used frequently throughout this document, not only when Symbian client/server architecture is discussed, but also to define the Bluetooth device roles. Within the context of this document, "client" refers to the device that searches for other devices and initializes the connection(s) (and becomes a master in the piconet). "Server" refers to the device that listens to and accepts the incoming connection request (and becomes a slave in the piconet). Note that in other Bluetooth literature these terms may be used the other way around to better reflect, for example, the game client/server roles in a multiplayer game scenario.

2. Overview of Bluetooth architecture in the S60 platform

Like many other communication technologies, Bluetooth is composed of a hierarchy of components, referred to as a stack, which is shown in Figure 1.

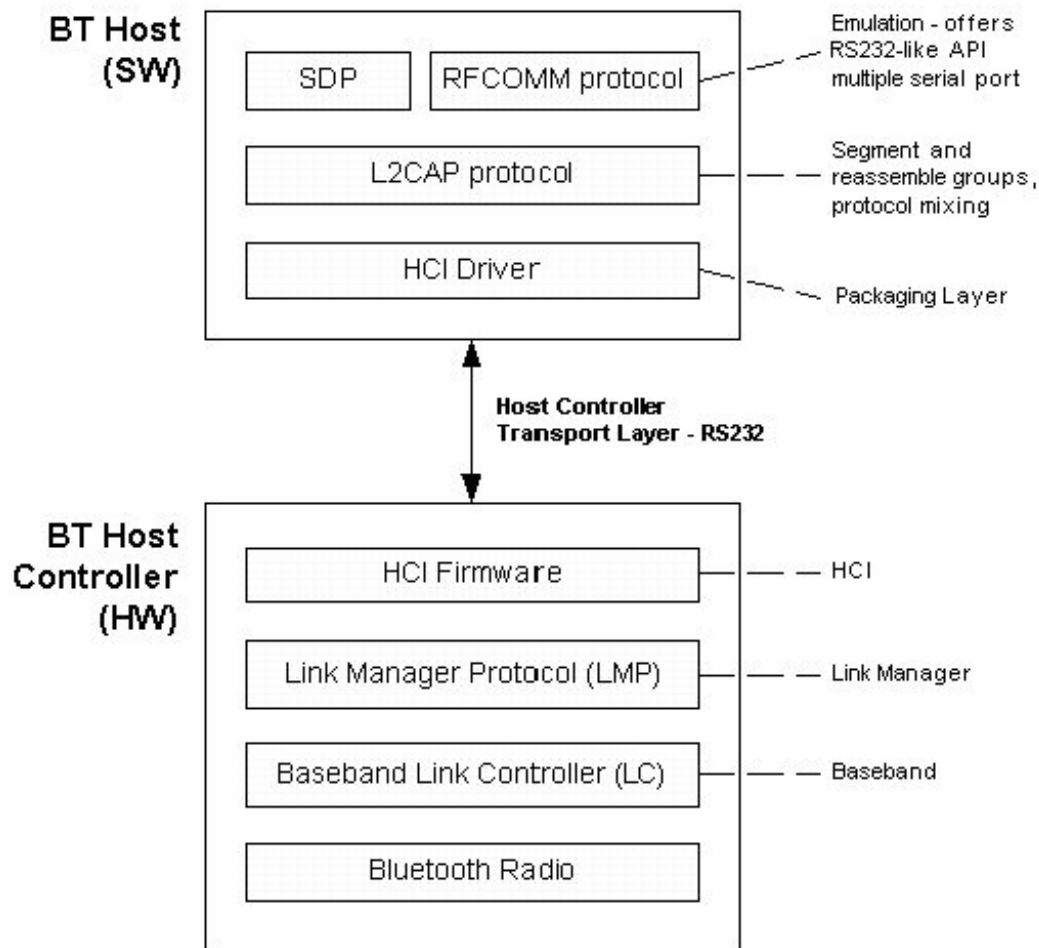


Figure 1: Bluetooth stack architecture

The Symbian OS Bluetooth APIs give applications access to RFCOMM, L2CAP, SDP, OBEX, and, to a limited extent, HCI.

The Bluetooth Host Controller components provide the lower level of the stack. The Host Controller components are normally implemented in the hardware. Applications do not have direct access to this layer.

Bluetooth Host components allow applications to send or receive data over a Bluetooth link or to configure the link:

- RFCOMM allows an application to treat a Bluetooth link in a way that is similar to communicating over a serial port. This is used to support legacy protocols.

- The Logical Link Control and Adaptation Protocol (L2CAP) allows finer-grained control of the link. It controls how multiple users of the link are multiplexed together, handles packet segmentation and reassembly, and conveys quality-of-service information.
- The Service Discovery Protocol (SDP) is used to locate and describe services provided by or available through a Bluetooth device. Applications typically use it when they are setting up communications to another Bluetooth device.
- The Host Controller Interface (HCI) driver packages the high-level components to communicate with the hardware. HCI commands provide a command interface to the baseband controller and link manager. These are provided through asynchronous I/O control (`ioctl`) commands on an L2CAP or RFCOMM socket, as there is no direct Symbian OS interface to the HCI. `Ioctl`s are issued through `RSocket::Ioctl()`. For details, see the reference for the `KHCIxxx` constants, such as `KHCIAddSCOCConnIoctl`.

2.1 Bluetooth API blocks

The Bluetooth APIs can be used from the following modules:

- Bluetooth Sockets: Encapsulate access to L2CAP and RFCOMM through a TCP/IP-like socket interface.
- Bluetooth Service Discovery Database: Encapsulates one side of the SDP. A local service uses it to record its attributes, so that remote devices may discover its presence and determine if it can be used.
- Bluetooth Service Discovery Agent: Encapsulates the other side of the SDP. It allows the user to discover which services are available on a remote device, as well as the attributes of those services.
- Bluetooth Security Manager: Enables services to set appropriate security requirements, which incoming connections must meet.
- Bluetooth UI (also called Bluetooth Device Selection UI): Provides an API by which a dialog asking users for device selection information can be called.

Figure 2 shows how the different Bluetooth APIs are related. Note that the Bluetooth Sockets API is the fundamental API upon which the other APIs rely to perform communications with other devices.

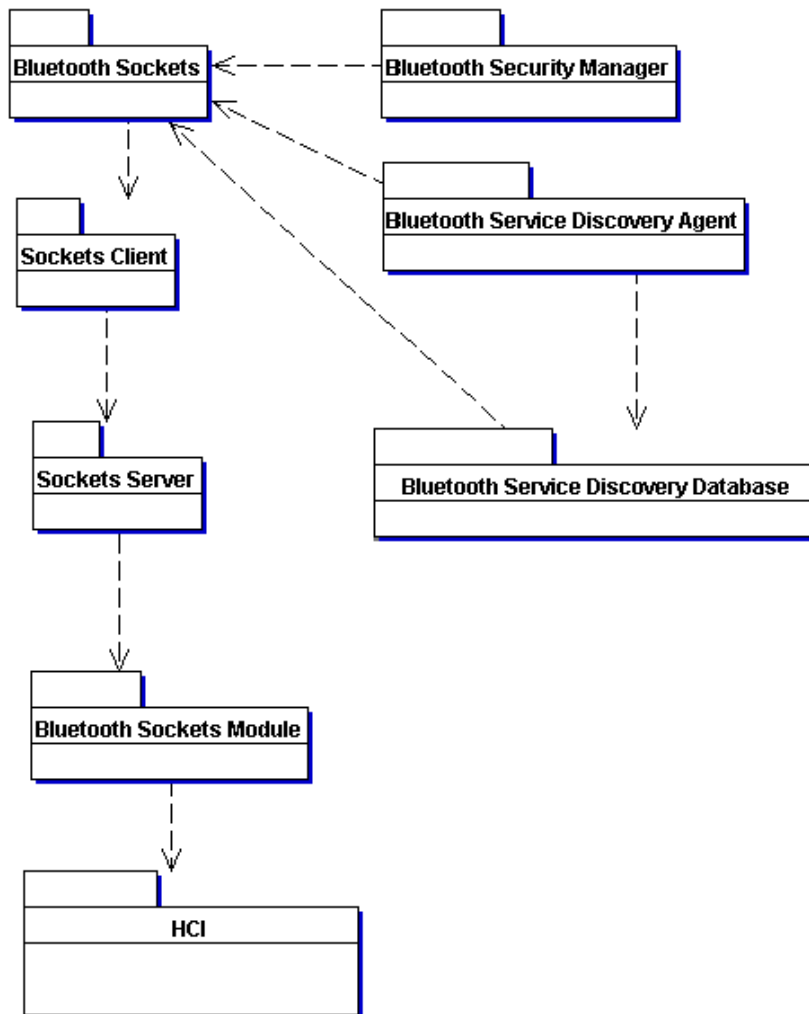


Figure 2: Architectural relationships between Symbian Bluetooth API modules

In addition to these general APIs there are a couple of S60-specific Bluetooth-related APIs. The Bluetooth Notifier API was introduced in S60 2nd Edition, Feature Pack 1. It was declared earlier in `btnotif.h` and changed to `btnotifierAPI.h` in S60 3rd Edition. It contains only a constant (notifier ID) for Bluetooth power mode setting. The Send UI API offers a high-level API for sending different types of messages, including Bluetooth (see Section 7.6, “Easy way of using OBEX with Send UI,” for a discussion of the Send UI API).

2.1.1 Bluetooth sockets

In Symbian OS, Bluetooth sockets are used to discover other Bluetooth devices and to read and write data over Bluetooth. The Bluetooth Sockets API (which includes the `RSocket` and `CBluetoothSocket` classes) supports communication over both the L2CAP and RFCOMM layers of the Bluetooth protocol suite. The API is based on the Sockets Client Side API, which provides a standard API allowing a client to make a connection to a remote device. Alternatively, it also allows a device to act as a server and have a remote device connect to it. Once connected, the devices may send and receive data before they disconnect. The Bluetooth Sockets API adds appropriate types and constants that enable the Sockets API to be used with Bluetooth.

The API has five key concepts: socket address, remote device inquiry, RFCOMM commands and options, L2CAP commands, and HCI commands.

This document concentrates on how to implement a Bluetooth connection using RFCOMM. As such, it focuses on showing how to establish such a connection. Before using the Sockets API to establish a connection, a Bluetooth device must locate a suitable device to connect to. The Bluetooth Service Discovery Protocol (SDP) performs this task.

The SDP can be broken down into two main parts:

1. The discovery of devices and services within the local area;
2. The advertisement of services from the local device.

2.1.2 Bluetooth Service Discovery Database

The Bluetooth Service Discovery Database allows a local service to enter its properties into a local Bluetooth service database. Doing this enables remote Bluetooth devices to discover that the service is available.

The Service Discovery Database API is one of two APIs that allows a developer to use the Bluetooth Service Discovery Protocol. The other one, the Bluetooth Service Discovery Agent, enables a developer to discover the Bluetooth services and the attributes of those services that are available on a remote device.

Use of the Service Discovery Database is described in Chapter 4, "Service discovery."

2.1.3 Bluetooth Service Discovery Agent

The Bluetooth Service Discovery Agent enables a developer to discover the Bluetooth services and the attributes of those services that are available on a remote device.

The Service Discovery Agent API is one of two APIs that enables the developer to use the Bluetooth Service Discovery Protocol. The other one, the Bluetooth Service Discovery Database, enables a local service to enter its own properties into a local service database.

The Service Discovery Agent is discussed in more detail in Chapter 4, "Service discovery."

2.1.4 Bluetooth Security Manager

The Bluetooth Security Manager enables Bluetooth services to set appropriate security requirements, which connections to that service must fulfill. The security settings define whether authentication, authorization, and encryption are required or not. If the remote device requires authentication or encryption to connect, the stack will handle this transparently. The usual case is that the receiving device (slave) sets security.

Beginning in Symbian OS v8.0a (S60 2nd Edition, Feature Pack 2) there is a new API that is used for setting security for both incoming and outgoing connections. In earlier versions (until Symbian OS v7.0s, S60 2nd Edition, Feature Pack 1), the API only deals with incoming connections; for outgoing connections, the Bluetooth stack will not by default enforce any particular Bluetooth security. It may occasionally be necessary for the local application to insist on authenticating or encrypting the link. In this case, control operations (`ioctl`s) can be issued on a connected socket to authenticate or encrypt the link.

The APIs for new and old platforms are covered in further detail in Chapter 8, “Security and configurations.”

2.1.5 Bluetooth UI

The Bluetooth UI is used in the selection of a remote device. It is also referred to as the Bluetooth Device Selection UI. When there are several suitable remote devices, it is advisable to prompt the user to pick the one to which a connection should be made. A dialog for doing this is supplied. The dialog is not provided by a standard dialog class, as this would require a Bluetooth client to have its own UI, and for this to be brought to the foreground. Instead, clients request a background thread called the Notifier Server to create the dialog; the server can put a dialog over the UI of whatever application happens to have the foreground.

An example of how to use the Bluetooth UI is provided in Section 3.1, “Using the Bluetooth UI.”

2.2 Common Bluetooth data types

The *Bluetooth Specification* defines many standard data types, structures, and containers that are used throughout a standard Bluetooth transaction.

Further information on how the Bluetooth protocol works can be obtained from the Bluetooth Web site, www.bluetooth.com. The entire *Bluetooth Specification* can be obtained from [1].

Most of the data types and containers used in the Bluetooth protocol were designed to transport information using as little data as possible. This constraint produced a set of data types and containers that can seem unfamiliar at first.

This section introduces developers to the major data types, structures, and APIs provided by Symbian OS Bluetooth implementation. All of these objects are defined in the S60 SDK under the following header files:

- `bt_sock.h`
- `btdevice.h`
- `btextnotifiers.h`
- `btsdp.h`
- `bttypes.h`

Read the header documents for more information about the definitions presented below.

2.2.1 Bluetooth device address

A remote Bluetooth device can be identified with a specific 48-bit Bluetooth device address, which is represented in the Symbian OS API as a `TBTDevAddr` class. This class is defined in the `bttypes.h` header file.

2.2.2 Universally Unique Identifier (UUID)

UUIDs are used throughout the SDP and specification. The specification calls for the use of many different types of UUIDs. All UUIDs are handled by the `TUUID` class. It can handle full-length, 128-bit-long UUIDs, and their shorter representations. The `TUUID` class is defined in the `bttypes.h` header file.

2.2.3 Service record

A Bluetooth service is identified by a UUID, and each service is represented by a Bluetooth Service Record. The service record contains the UUID and a set of attributes providing information on that service. A service record class is not explicitly declared in the Symbian OS API. Instead, the API provides a Bluetooth SDP database to allow an application to advertise a service, as well as a set of classes that enables an application to search for available services and their related attributes.

2.2.4 Service class and profiles

Service classes are represented by UUIDs. They are used to help generalize the types of service provided by devices. For instance, the specification, which can be found at [1] <http://www.bluetooth.org> (registration needed), lists predefined numbers that represent a printer and another, more specific entry that defines a color printer. A profile of required service attributes is also provided with each type of service class.

2.2.5 Service record handle

All service records are listed in an SDP database. The SDP database identifies different service records by a 32-bit number, or a service record handle. A service record handle is represented by the `TsdpServRecordHandle` data type, which is declared in the `bt_sdp.h` header file.

2.2.6 Service attribute ID

Each service record contains a collection of attributes that are identified by an ID number. This number is represented by the `TsdpAttributeID` data type. This data type is not a class, but does in fact correspond to a 16-bit integer. The data type is defined in the `bt_sdp.h` header file. The Symbian OS SDP database uses the service record handle and the attribute ID to identify attributes and their values in the database. This is covered in more detail in Chapter 4, "Service discovery."

2.2.7 SDP database

Each Bluetooth device contains a local database of all the services it can provide. This database is represented in the Symbian OS API as a class `RSdpDatabase`. This class is defined in the `bt_sdp.h` header file. More information on how to use this class is provided in Section 4.1, "CSdpAgent and MSdpAgentNotifier classes."

2.2.8 Data elements

Table 1 shows how each of the Bluetooth data element types are represented in the Symbian OS API, their immediate parent or base class, and the header file location for the definitions. All of the data types shown in the table are declared in the `bt_sdp.h` header file.

All data element classes in the Symbian OS API are derived from a common base class `CSdpAttrValue`. With the exception of the `CSdpAttrValueDES` and `CSdpAttrValueDEA` classes, all other classes are derived directly from the `CSdpAttrValue` base class.

Bluetooth data element type	Symbian OS representation
Nil	<code>CSdpAttrValueNIL</code>
Unsigned Integer	<code>CsdpAttrValueUInt</code>
Signed Twos-Complement Integer	<code>CsdpAttrValueInt</code>
UUID (Universally Unique Identifier)	<code>CSdpAttrValueUUID</code>
Text String	<code>CsdpAttrValueString</code>
Boolean	<code>CsdpAttrValueBoolean</code>
Data Element Sequence (list)	<code>CSdpAttrValueDES</code>
Data Element Alternative (list)	<code>CSdpAttrValueDEA</code>
URL	<code>CSdpAttrValueURL</code>

Table 1: Bluetooth data types in Symbian OS

The data types are defined in the `btsdp.h` header file.

2.2.9 CSdpAttrValue data type values

With the exception of lists, each of the data element types defined is fairly straightforward to use. All data types have the same basic API. The functions for the `CSdpAttrValueInt` are described here:

- `NewIntL` - This constructor accepts any undefined quantity as a byte buffer for the data type.
- `Type` - This function returns an enumerated value representing the type of data stored in the class.
- `DataSize` - Returns the size in bytes that this particular data element would consume.
- `Int` - Provides access to the data value itself.
- `DoesIntFit` - Returns true if the data type this class represents fits into an integer word size.
- `Des` - Returns a pointer to the byte buffer containing the actual data handled by the class.

2.2.10 Service search patterns

Service search patterns have their own class in the Symbian OS API. The `CSdpSearchPattern` class is defined in the `btsdp.h` header file. `CSdpSearchPattern` is used as a pattern search object by the operating system.

When studying the public interface definition of the `CSdpSearchPattern` class, it is possible to see that it provides all the necessary interfaces to create a list of `TUUIDS`. These are then used to search for available services in the local area.

2.2.11 Attribute search patterns

The Symbian OS API also provides a class for creating a list of attribute IDs in a way that is similar to the service search patterns. `CSdpAttrIdMatchList` is used to issue a request for attribute values from a remote SDP database. Its public interface is very similar to the service search pattern's API explained previously.

2.2.12 Attribute ranges

One noticeable difference between `CSdpSearchPattern` and `CSdpAttrIdMatchList` is that `CSdpAttrIdMatchList` accepts a `TAttrRange` structure. This structure allows a client application to specify a range of attribute IDs it is interested in. The `TAttrRange` structure is defined in the `bt_sdp.h` header file.

2.3 Bluetooth API v2 architecture

The original Bluetooth API architecture has experienced major improvements in Symbian OS v8.0a (S60 2nd Edition, Feature Pack 2). They include:

- New methods for setting up a listening socket (see Section 6.5, "Listening sockets," for details);
- New method for setting security (see Section 8.1.1, "Service security in Bluetooth API v2 architecture," for details);
- Stack/hardware settings that are now set by the Publish and Subscribe API;
- `CBluetoothSocket`, which provides better Bluetooth functionality than the previously used `RSocket`.



Note: The `__BLUETOOTH_API_V2__` macro is not declared in the S60 platform until S60 3rd Edition (`symbian_os_v9.1.hrh`). In practice, this means that developers have to create different projects for releases supporting the old and new Bluetooth architecture. In addition, they need to create different builds for S60 3rd Edition.

However, developers can use the S60 Identification Code of S60 2nd Edition, Feature Pack 2, in the PKG file to indicate the compatibility at installation time. For further information on identification codes, see the [S60 Platform: Identification Codes](#) document at www.forum.nokia.com [6].

More changes are described throughout this document and summarized in Appendix A.

2.4 Platform security — effects on Bluetooth application development

The biggest change in Symbian OS v9.1 (S60 3rd Edition) is the platform security concept. Its main building blocks are Capabilities (set of privileges to applications), Data Caging (secure storage of data), Secure Interprocess Communication (IPC), and memory management. Recognizers, notifiers, and converter plug-ins are implemented as ECOM plug-ins. To allow applications to have different capabilities, applications now have to be defined as .EXEs (they were previously polymorphic DLLs with extension `.app`). Platform security also requires a number of changes to the application architecture.

In addition to general changes in applications, Bluetooth developers need to check the required capabilities for certain actions. Here is a list of capabilities that might be needed for certain Bluetooth-related actions. For more information, see the “capability report” in the S60 3rd Edition SDK Help.

Most Bluetooth applications should work with the user capabilities that can be granted with the user's confirmation so they are available for self-signed applications. Some functions in the `CBluetoothPhysicalLinks` class require `NetworkControl` capability that requires the developer to fill in the Capability Request Form and acceptance from the platform manufacturer (see [Symbian Signed](#) for more information). Table 2 lists the most-needed capabilities in Bluetooth applications.

Capability	Description	Classification
LocalServices	Grants access to local network services that usually do not incur a cost.	User-grantable
NetworkServices	Grants access to remote network services that may incur a cost. This capability is granted on a one-shot basis if not previously authorized.	User-grantable
ReadUserData	Grants read access to data that is confidential to the mobile phone user (subject to confirmation).	User-grantable
WriteUserData	Grants write access to data that is confidential to the mobile phone user (subject to confirmation).	User-grantable
ReadDeviceData	Grants read access to sensitive system data.	Symbian Signed (declarative justification required)
WriteDeviceData	Grants write access to sensitive system data.	Symbian Signed (declarative justification required)
NetworkControl	Grants access or modification rights to network protocol controls.	Symbian Signed (Capability Request Form and Platform approval required)

Table 2: Mostly needed capabilities in Bluetooth applications

Table 3 lists the capabilities that may be needed when using the specified Bluetooth components.

Component	Capabilities	Description
BLUETOOTH_SDP	LocalServices	Usage of <code>CSdpAgent</code> or <code>RSdpDatabase</code> requires <code>LocalServices</code> capability.
In BLUETOOTH_USER <code>CBluetoothPhysicalLinks</code> <code>CBluetoothSocket</code> and <code>CBluetoothSynchronousLink</code>	<code>NetworkControl</code> <code>LocalServices</code>	<code>LocalServices</code> is also required when using <code>CBluetoothSocket</code> . <code>CBluetoothPhysicalLinks</code> requires <code>NetworkControl</code> and may therefore be unusable for developers.
BLUETOOTH_MANAGER	<code>LocalServices</code> <code>WriteDeviceData</code> <code>ReadDeviceData</code>	For creating view, modifying devices in registry, etc., different capabilities will be required.
<code>RSocket::Ioctl</code>	<code>NetworkServices</code>	Required for <code>KIoctlGetMsgId</code> and <code>KIoctlGetLastSendError</code> commands.

Table 3: Capabilities needed with specified Bluetooth components

To put it together, platform security should not affect development of regular Bluetooth applications that only use high-level APIs. However, if the application needs to use low-level interfaces (such as modifying the remote device data), even manufacturer domain capabilities may be needed.

3. Device discovery

In Symbian OS there are two main ways to search for devices. First, a client application may issue a query to all devices within range and handle each response in turn. Alternatively, an application could use the Bluetooth UI. This UI plug-in will automatically issue the query, handle the responses, and prompt users with a dialog box. The dialog box allows users to select the device they want to use. Another possibility is to use the `RHostResolver` class. These device discovery methods are discussed next.

3.1 Using the Bluetooth UI

The Bluetooth UI does all the work of searching for available Bluetooth devices and presenting them to users. It is provided as a plug-in to the `RNotifier` class. The `RNotifier` class is a generic class that allows the use of several plug-in UI elements provided by the operating system. It is defined in the `e32std.h` header file.

The `RNotifier` class is designed to be used with most types of client applications. `RNotifier` provides access to a background thread that will show the selection dialog on the screen. This implementation method means that is not necessary for the client application to have a GUI interface; accordingly, this means that the Bluetooth Device Selection can be used in DLLs.

Since the `RNotifier` class is a generic class, it is designed to be used with different types of dialogs. A unique ID represents each type of dialog. The Bluetooth UI dialog for device selection is represented by the `KDeviceSelectionNotifierUid` constant value. It is defined in the `bttextnotifiers.h` header file.

The device selection thread uses a set of classes to transfer data and settings to and from the client application. These are briefly explained in the following paragraphs.

The `TBTDeviceSelectionParams` class allows a client application to pass initial parameters to the selection process.

The client application can set the device class to search for. By setting this value with the `SetDeviceClass` function, the client application can limit the number and type of devices that will respond to the search request (notice the contrast to limited inquiry in Section 3.2.3, "Setting the Inquiry Access Code (IAC)"), and thus those that are displayed to the user.

It is also possible for the client application to set the UUID of the service it requires. This again will constrain the resulting list of devices presented to the user. This can be done with the `SetUUID` function.

To search for all available devices, no matter what type of service they provide, a client application would leave the `TBTDeviceSelectionParams` object unaltered with its default values.

After the user has selected a preferred device, the background thread will complete. The user's selection is presented to the client application via the `TBTDeviceResponseParams` class.

The `IsValidxxxx` functions are used to ensure that the relevant information has been set in the class; for instance, if the `IsValidBDAddr` function returns true, the client application can expect that the Bluetooth address of the selected device has been set in the `TBTDeviceResponseParams` object. If the `IsValidBDAddr` function returns false, then this data is not present. This will happen if an error has occurred.

The client application can obtain the Bluetooth device address of the selected device via the `BDAddr` function, and the device name via the `DeviceName` function.

As stated previously, the `RNotifier` class provides access to a background thread that actually performs all of the hard work of searching for devices and throwing the dialog. This is implemented as a server process. Package buffers are required to communicate data over the client/server boundary. The Symbian OS API provides a set of package buffers that wrap the data types defined before. The package buffers are as follows:

- `TBTDeviceSelectionParamsPckg`
- `TBTDeviceSelectionParams`
- `TBTDeviceResponseParamsPckg`
- `TBTDeviceResponseParams`

The Bluetooth OBEX example [4], available from Forum Nokia at www.forum.nokia.com, implements the Bluetooth Device Selection Notifier. The following codes are taken from `BtServiceSearcher.cpp`.

As noted previously, `RNotifier` provides access to a server-side process. Before `RNotifier` can be used, the client application must first connect to the server.

```
RNotifier iDeviceSelector; //in btservicesearcher.h
User::LeaveIfError(iDeviceSelector.Connect());
```

A `TBTDeviceSelectionParamsPckg` object is created; this object contains an instance of the `TBTDeviceSelectionParams` class. Note that this class is left initialized with the default values. A Bluetooth Device Selection dialog based on this selection parameter should give the user a list of all discoverable devices in range.

`RNotifier` is designed to use an Active Object's `TRequestStatus` to send a notification when the user makes a selection. In the OBEX example, the `CObjectExchangeClient::ConnectL` function calls the `SelectDeviceByDiscoveryL` function passing its `iStatus` member as an argument. After the device selection is done, `CObjectExchangeClient::RunL` is called. The `CObjectExchangeClient` class is a complicated state machine, so it is not discussed in detail here.

`RNotifier` is called to display the Bluetooth device selection dialog. At this point, `RNotifier` executes the code to search for all available devices and presents this list to the user. The first parameter is `TRequestStatus`; the second parameter is the constant integer that represents the Bluetooth device selection dialog; the third parameter is the Selection Filter (not supported by the S60 platform); and the final parameter is the response parameter, `TBTDeviceResponseParamsPckg`, which is populated when the user makes a selection. This is an asynchronous call: the current thread is not paused and continues to the next statement.

```
iDeviceSelector.StartNotifierAndGetResponse(
    aObserverRequestStatus,
    KDeviceSelectionNotifierUid,
    selectionFilter,
    iResponse );
```

The `TRequestStatus` object is also used to return any error states encountered during the execution of the device selection code.

Once the client application has finished with the `RNotifier` object, it is shut down. This is accomplished by telling `RNotifier` to unload the Bluetooth device selection dialog code and then to close its connection with the server. If the `RNotifier`'s server is not being used by any other application, it is unloaded from the system as well, thus freeing resources.

```

if ( iIsDeviceSelectorConnected )
{
    iDeviceSelector.CancelNotifier(KDeviceSelectionNotifierUid);
    iDeviceSelector.Close();
}

```

Further information on the Bluetooth device selection dialog is available in the SDK Help in the S60 SDK.

3.1.1 Limitations of the device selection notifier plug-in

When using the notifier, a developer has to initiate the device discovery by launching the device selection notifier and selecting the remote device. This means that no programmatic inquiries can be made with this plug-in. Point-to-multipoint connections (as in the Bluetooth PMP example [5]) are supported from S60 2nd Edition, Feature Pack 3 onwards with the restriction that only one connection at a time can be established. Also, the UI component of the plug-in may not fit into the look and feel of an application if the developer has designed a tailored menu style.

For advanced options, the developer will have to use the `RHostResolver` class for the inquiry (see Section 3.2, "Running a device discovery with `RHostResolver`").

3.2 Running a device discovery with `RHostResolver`

Address and name inquiries are performed through the generic Symbian OS sockets class `RHostResolver`. `TInquirySockAddr` is provided for such inquiries. It is a Bluetooth sockets address class that encapsulates the Bluetooth address, Inquiry Access Code, and service and device classes.

`RHostResolver` class is defined in the `es_sock.h` header file. Its public interface is as follows:

```

class RHostResolver : public RSubSessionBase
{
public:
    IMPORT_C TInt Open(RSocketServ& aSocketServer, TUint
anAddrFamily, TUint aProtocol);
    IMPORT_C TInt Open(RSocketServ& aSocketServer, TUint
anAddrFamily, TUint aProtocol, RConnection& aConnection);
    IMPORT_C void GetByName(const TDesC& aName, TNameEntry&
aResult, TRequestStatus& aStatus);
    IMPORT_C TInt GetByName(const TDesC& aName, TNameEntry&
aResult);
    IMPORT_C void Next(TNameEntry& aResult, TRequestStatus&
aStatus);
    IMPORT_C TInt Next(TNameEntry& aResult);
    IMPORT_C void GetByAddress(const TSockAddr&
anAddr, TNameEntry& aResult, TRequestStatus& aStatus);
    IMPORT_C TInt GetByAddress(const TSockAddr&
anAddr, TNameEntry& aResult);
    IMPORT_C TInt GetHostName(TDes& aName);
    IMPORT_C void GetHostName(TDes& aName, TRequestStatus
&aStatus);
    IMPORT_C TInt SetHostName(const TDesC& aName);
    IMPORT_C void Close();
    IMPORT_C void Cancel();

    IMPORT_C void Query(const TDesC8& aQuery, TDes8& aResult,
TRequestStatus& aStatus);
    IMPORT_C TInt Query(const TDesC8& aQuery, TDes8& aResult);

```

```

    IMPORT_C void QueryGetNext(TDes8& aResult, TRequestStatus&
aStatus);
    IMPORT_C TInt QueryGetNext(TDes8& aResult);

private:
    };

```

3.2.1 Getting the address of a remote device

To inquire for the addresses of remote devices using `RHostResolver`:

1. Connect to the sockets server (`RSocketServ`) and select the protocol to be used with `RSocketServ::FindProtocol()`. Address and name queries are supplied by the Bluetooth stack's `BTLinkManager`.
2. Create and initialize an `RHostResolver` object.
3. Set the parameter for the inquiry. For address inquiries, the `KHostResInquiry` flag must be set through `TInquirySockAddr::SetAction()`.

The query can then be started with `RHostResolver::GetByAddress()`.

4. When `GetByAddress()` completes, it fills in a `TNameEntry` object with the address and class of the first device found (or is undefined if no device was found).
5. To get all the devices discovered, call `RHostResolver::Next()` repeatedly until `KerrHostResNoMoreResults` is returned.

This is demonstrated in the Bluetooth PMP example [5].

3.2.2 Getting the name of a remote device

The procedure is the same as for the address query but the action flag of a `TInquirySockAddr` has to be set to `KHostResName`.

The name is returned in the member accessed through the `TNameEntry`.

To do a simultaneous address and name inquiry, use `SetAction(KHostResName|KHostResInquiry)`.

3.2.3 Setting the Inquiry Access Code (IAC)

In the *Bluetooth Specification* there are two different inquiry access codes for different use cases. The default case is that the inquiring device finds all the devices that are discoverable in the neighborhood. In this case the searching device is inquiring with the Generic Inquiry Access Code (GIAC). However, this can take a considerably long time to complete.

Therefore, developers can also set the scanning devices into Limited-Discoverable mode. When making a device inquiry with Limited Inquiry Access Code (LIAC), they can find these devices because the other normal discoverable devices are filtered out, and they only find the devices that are in the limited-discoverable mode.

LIAC is useful in cases where an application is running on two or more devices and connection is desired to these devices only.

Devices scanning with LIAC can be found by devices inquiring with LIAC or GIAC (the priority is in LIAC inquiries).

Devices inquiring with LIAC can only find devices scanning with LIAC.

For an application to fully take advantage of the faster LIAC inquiry, it is crucial that the device performing the device inquiry is also set to LIAC mode.

`TIquirySockAddr::SetIAC()` is the method for setting the Bluetooth Inquiry Access Code. `KGIAC` is used for the generic inquiry and `KLIAC` for the limited inquiry.

There are two types of inquiries as mentioned above. A limited discoverable device responds only to limited inquiries, which makes limited inquiries a lot faster.

Limited inquiry is set using the Symbian OS Publish and Subscribe API as follows:

```
socket.Ioctl(KHCIWriteDiscoverabilityIoctl,
status, &mode(=KLIAC), KSolBtHCI); returns -5 in V2

//In V2:
//first define the property:
RProperty::Define(KPropertyUidBluetoothControlCategory,
KPropertyKeyBluetoothLimitedDiscoverable,
RProperty::EByteArray);
//KPropertyKeyBluetoothGetLimitedDiscoverableStatus
//in S60 3rd Edition!

TPkgBuf<TUint32> type = KLIAC; // or KGIAC
int error =
RProperty::Set(KPropertyUidBluetoothControlCategory,
KPropertyKeyBluetoothLimitedDiscoverable, type);

//OR directly like this:
int error =
RProperty::Set(KPropertyUidBluetoothControlCategory,
KPropertyKeyBluetoothLimitedDiscoverable, (mode ==
KLIAC)?ETrue:EFalse);
```

3.2.4 Device discovery in the Bluetooth point-to-multipoint example

The following code excerpts are taken from the Bluetooth PMP example [5]. The example has been written for the S60 platform but the parts related to the Bluetooth functionality are applicable to other platforms as well.

In the PMP example the `CBluetoothPMPEngine` manages listening operations, discoveries, connections, data sending, and disconnections. The Socket server session is opened in the construction:

```
void CBluetoothPMPEngine::ConstructL()
{
    User::LeaveIfError(iSocketServ.Connect());
}
```

`CBluetoothPMPEngine::DiscoverDevicesL()` calls the `CDeviceDiscoverer::DiscoverDevicesL()` function when the user has chosen to start the discovery. Any found devices are placed in the created `aDevDataList`:

```
void CDeviceDiscoverer::DiscoverDevicesL(TDeviceDataList
*aDevDataList)
{
```

```
iDiscoveredDeviceCount=0;
```

Existing device data is cleared:

```
iDevDataList=aDevDataList;
iDevDataList->Reset();
```

Protocol for the discovery is loaded:

```
TProtocolDesc pdesc;
User::LeaveIfError(iSocketServ->FindProtocol
    (_L("BTLinkManager"), pdesc));
```

The Host Resolver is initialized:

```
User::LeaveIfError(iResolver.Open(*iSocketServ,
    pdesc.iAddrFamily, pdesc.iProtocol));
```

Device discovery is started by invoking a remote address lookup. The Generic Inquiry Access Code is selected by using a `SetIAC(KGIAC)` function. Simultaneous address and name inquiry is selected by using the `SetAction(KHostResInquiry|KHostResName)` function. By adding a `KHostResIgnoreCache` parameter, you do not retrieve the names from the cache but make a genuine name discovery:

```
iAddr.SetIAC(KGIAC);
iAddr.SetAction(KHostResInquiry|KHostResName|
    KHostResIgnoreCache);
iResolver.GetByAddress(iAddr, iEntry, iStatus);
SetActive();
}
```

```
void CDeviceDiscoverer::RunL()
{
```

The discovery `RHostResolver.GetByAddress(..)` is completed. Call the `CBluetoothPMPEExampleEngine::HandleDeviceDiscoveryCompleteL()` function:

```
if ( iStatus != KErrNone )
{
    iObserver->HandleDeviceDiscoveryCompleteL();
}
```

If there are still devices left, add the name and the address of the newest found device to the list:

```
else
{
```

New device data entry:

```
TDeviceData *devData = new (ELeave) TDeviceData();
devData->iDeviceName = iEntry().iName;
devData->iDeviceAddr =
    static_cast<TBTSockAddr>(iEntry().iAddr).BTAddr();
devData->iDeviceServicePort = 0;
```

Add the device data entry and increase the amount of discovered devices:

```
iDevDataList->Append(devData);
iDiscoveredDeviceCount++;
```

Get the next discovered device:

```
iResolver.Next(iEntry, iStatus);
```

Wait for the resolver to complete:

```
        SetActive();
    }
}
```

Cancel and close the resolver:

```
void CDeviceDiscoverer::DoCancel()
{
    iResolver.Close();
}
```

A callback from the device discoverer indicates that the device discovery has completed:

```
void
CBluetoothPMPEngine::HandleDeviceDiscoveryCompleteL()
{
}
```

Iterate and display found devices:

```
if ( iDevDataList.Count() > 0 )
{
    ShowMessageL( L("Found devices:\n"), ETrue;
    for (TInt idx=0; idx<(iDevDataList.Count()); idx++)
    {
        TDeviceData *dev = iDevDataList[idx];
        TBuf<40> name;
        name = dev->iDeviceName;
        name.Append( L("\n"));
        ShowMessageL(name, EFalse);
    }
}
else
{
}
```

If no devices were found:

```
    ShowMessageL( L("\nNo devices found!"), EFalse);
}
}
```

3.2.5 Notes about the RHostResolver discovery

Using the `RHostResolver` enables developers to design advanced discovery and device selection methods for their applications. They can run programmatic discoveries (periodical discoveries running in the background, for example) and connect programmatically to the discovered devices.

Point-to-multipoint connections can be established and new connections added while the device is already connected. The typical case would be to establish all connections from one device (master).

From S60 2nd Edition, Feature Pack 3 onwards, scatternets are also supported (this means that a device that is slave in one piconet can establish connections to other devices and be a master in another piconet).

Developers can also design the UI components related to the discovery and the device selection list according to the style of their application.

The drawback is that they have to implement pretty much manually (gathering the found devices in the database and showing them to the user if required).

Thus, using `RNotifier` is recommended whenever the advanced capabilities of the `RHostResolver` method are not needed.

Finally, before running device discovery, check if Bluetooth is set on (and prompt the user to switch it on, if necessary). See Section 8.4.1, "Asking the user to turn Bluetooth power on," for details.

4. Service discovery

Once the list of available devices and their addresses has been generated, the client application can inquire a given device for the services it requires (device service request). When a device with a suitable service has been identified, a client application will usually also query the service for more information. This involves requesting the available attributes of the identified service (service attribute request). For service discoveries, Symbian OS provides a Service Discovery Agent API.

This chapter introduces several new parts of the Symbian OS Bluetooth API. It also provides a more detailed description of some of the definitions introduced in Chapter 2, "Overview of Bluetooth architecture in the S60 platform." Note, however, that the main purpose of this document is to describe how the Symbian OS API makes Bluetooth functionality available for application developers. For information on how Bluetooth protocols work, see www.bluetooth.org.

4.1 CSdpAgent and MSdpAgentNotifier classes

Searching for available services in a Bluetooth piconet can be a time-consuming task. Symbian OS provides an asynchronous API to perform service searches. Since this API is asynchronous, a client application must provide a set of callback notification functions. These functions are referred to as observer functions.

The Symbian OS Bluetooth API provides the `MSdpAgentNotifier` observer class that defines a set of pure virtual observer functions that need to be implemented by a client application.

All search requests are submitted to the `CSdpAgent` class. This class is defined in the `btsdp.h` header file. The `CSdpAgent`'s public interface is shown in the following code:

```
class CSdpAgent : public CBase
{
public:
IMPORT_C static CSdpAgent* NewL(MSdpAgentNotifier& aNotifier,
    const TBTDevAddr& aDevAddr);
IMPORT_C static CSdpAgent* NewLC(MSdpAgentNotifier& aNotifier,
    const TBTDevAddr& aDevAddr);
IMPORT_C ~CSdpAgent();
IMPORT_C void SetRecordFilterL(const CSdpSearchPattern&
    aUUIDFilter);
IMPORT_C void SetAttributePredictorListL(const
    CSdpAttrIdMatchList&
    aMatchList);
IMPORT_C void NextRecordRequestL();
IMPORT_C void AttributeRequestL(TSdpServRecordHandle aHandle,
    TSdpAttributeID aAttrID);
IMPORT_C void AttributeRequestL(TSdpServRecordHandle aHandle,
    const CSdpAttrIdMatchList& aMatchList);
IMPORT_C void AttributeRequestL(MSdpElementBuilder* aBuilder,
    TSdpServRecordHandle aHandle, TSdpAttributeID aAttrID);
IMPORT_C void AttributeRequestL(MSdpElementBuilder* aBuilder,
    TSdpServRecordHandle aHandle, const CSdpAttrIdMatchList&
    aMatchList);
IMPORT_C void Cancel();
...
};
```

The `CSdpAgent` must be created by specifying an `MSdpAgentNotifier` observer object and a Bluetooth device address. This is the address of the device that the client

application wishes to query. The observer class `MSdpAgentNotifier` is defined in the `btsdp.h` header file.

4.1.1 Creating the instance of `CSdpAgent`

Before a client application can start searching the services offered by a remote device, it must first create an instance of a class that implements the `MSdpAgentNotifier` interface.

The `CSdpSearchPattern` class introduced in Chapter 2, "Overview of Bluetooth architecture in the S60 platform," is used to specify a set of service IDs. Any service records held in the remote SDP database that contain any one of these IDs will be returned during the search process.

Creating an instance of the `CSdpAgent` class is only possible when a client application has both identified the address of a remote Bluetooth device and created an object that implements the `MSdpAgentNotifier` interface.

A client application can use the `CSdpAgent`'s `SetRecordFilterL` function to set the list of services it is interested in. This function accepts a reference to a `CSdpSearchPattern` object.

4.1.2 Running the service search

To start the search process, the client application needs only to call the `CSdpAgent`'s `NextRecordRequestL` function. This initiates the asynchronous search process. The operating system will inform the client application of its progress using the `MSdpAgentNotifier` class.

The `MSdpAgentNotifier::NextRecordRequestComplete` function will be called when the original service search request completes. Its parameters specify an error code (`aError`), a service record handle (`aHandle`), and an integer value (`aTotalRecordsCount`). The error code will contain one of the standard Symbian error codes, for example, `KErrNone`. This indicates the success of the search operation. The service record handle will be populated with the first record that matches the search filter parameters. Finally, the integer value (`aTotalRecordsCount`) saves the number of records found that match the filter. Subsequent calls to `CSdpAgent::NextRecordRequestL` result in the operating system calling the `NextRecordRequestComplete` function with the next matching service record handle and an updated integer value (`aTotalRecordsCount`).

4.1.3 Running the attribute search

When the `NextRecordRequestComplete` function is triggered, the client application would typically interrogate the remote device for more information about the service record it is advertising.

The `CSdpAgent`'s `AttributeRequestL` function initiates the attribute search process. The overloaded `AttributeRequestL` function can accept (and will search for) either a single attribute ID or a collection of IDs.

To specify a collection of IDs, the client application must use the `CSdpAttrIdMatchList` class. As stated in Chapter 2, "Overview of Bluetooth architecture in the S60 platform," this class will allow the client application to specify ranges of attribute IDs.

The `AttributeRequestL` function, which starts the attribute search, is also asynchronous. The operating system will call the

`MSdpAgentNotifier::AttributeRequestResult` function once for each attribute that matches the search criteria.

Once all attributes matching the search parameters have been returned to the client application, the observer function `MSdpAgentNotifier::AttributeRequestComplete` will be invoked.

4.1.4 Bluetooth Chat example

The S60 3rd Edition SDK provides a Bluetooth Chat example. In the Chat example, the `CChatBtServiceSearcher` actually implements the `MSdpAgentNotifier` interface. This is defined in the `series60Ex\Chat\inc\ChatBtServiceSearcher.h` header file.

`CChatBtServiceSearcher` both implements the `MSdpAgentNotifier` interface and encapsulates the `CSdpAgent` object. `CChatBtServiceSearcher` is responsible for both issuing the search requests and handling the operating system responses.

Implementation of the `CChatBtServiceSearcher` class can be found in the `series60Ex\Chat\src\ChatBtServiceSearcher.cpp` file. This code forms the basis of the following discussion.

The `FindServiceL` function illustrates how to initiate the service search process.

```
void CChatBtServiceSearcher::FindServiceL( TRequestStatus&
aObserverRequestStatus )
{
    if ( !iResponse().IsValidBDAddr() )
    {
        User::Leave( KErrNotFound );
    }
    iHasFoundService = EFalse;

    // delete any existing agent and search pattern
    delete iSdpSearchPattern;
    iSdpSearchPattern = NULL;

    delete iAgent;
    iAgent = NULL;

    iAgent = CSdpAgent::NewL( *this, BTDevAddr() );

    iSdpSearchPattern = CSdpSearchPattern::NewL();

    iSdpSearchPattern->AddL( ServiceClass() );
    // return code is the position in the list that the UUID is
    // inserted at and is intentionally ignored

    iAgent->SetRecordFilterL( *iSdpSearchPattern );

    iStatusObserver = &aObserverRequestStatus;

    iAgent->NextRecordRequestL();
}
```

The `CChatBtServiceSearcher::NextRecordRequestComplete` function handles the service record request responses.

```
void CChatBtServiceSearcher::NextRecordRequestComplete(TInt
aError, TSdpServRecordHandle aHandle, TInt aTotalRecordsCount)
{
```

It first checks to ensure that the application has not already received all the matching records, and that the request has completed without an error.

```

    if ( aError == KErrEof )
    {
        Finished();
        return;
    }

    if ( aError != KErrNone )
    {
        iLog.LogL( KErrNRRCErr, aError );
        Finished( aError );
        return;
    }

    if ( aTotalRecordsCount == 0 )
    {
        HBufC* errNRRCNoRecords = StringLoader
            ::LoadLC ( R_CHAT_ERR_NRRC_NO_RECORDS );
        iLog.LogL( *errNRRCNoRecords );
        CleanupStack::PopAndDestroy ( errNRRCNoRecords );
        Finished( KErrNotFound );
        return;
    }

    // Request its attributes

```

If there are no errors and no service record to process, the example application issues an attribute request to obtain more information about the service that is advertised.

```

iAgent->AttributeRequestL( aHandle,
KSdpAttrIdProtocolDescriptorList );

```

CChatBtServiceSearcher::AttributeRequestResult handles the results of the original attribute request. It parses the attribute values to see if there's a suitable service found.

```

void CChatBtServiceSearcher::AttributeRequestResultL(
    TSdpServRecordHandle /*aHandle*/,
    TSdpAttributeID aAttrID,
    CSdpAttrValue* aAttrValue )
{
    _ASSERT_ALWAYS( aAttrID ==
KSdpAttrIdProtocolDescriptorList,
        User::Leave( KErrNotFound ) );
    TChatSdpAttributeParser parser( ProtocolList(), *this );

    // Validate the attribute value, and extract
    //the RFCOMM channel
    aAttrValue->AcceptVisitorL( parser );

    if ( parser.HasFinished() )
    {
        // Found a suitable record so change state
        iHasFoundService = ETrue;
    }
}

void CChatBtServiceSearcher::AttributeRequestComplete(
    TSdpServRecordHandle aHandle,
    TInt aError )
{
    TRAPD(error,AttributeRequestCompleteL( aHandle, aError ));
    if ( error != KErrNone )

```

```
{  
    Panic (EChatBtServiceSearcherAttributeRequestComplete);  
}
```

5. Discoverability and advertising

For Bluetooth applications to function they need to advertise their services so that the other party can discover them.

5.1 Advertising a service

The following section describes how a client application can advertise its own set of Bluetooth services. This document will reference code from the Chat example [3], which is available in S60 3rd Edition SDKs. There are also useful Bluetooth examples in the S60 2nd Edition SDK, such as the Bluetooth Advertiser example.

5.1.1 The SDP database

All Bluetooth services are represented by the service record. As mentioned in Chapter 2, "Overview of Bluetooth architecture in the S60 platform," the Symbian OS API does not have an individual Service Record class. Instead, the SDP database directly controls access to service records and their attributes.

The SDP database of a Symbian OS device is responsible for publicizing all of the services that the device can offer. This means that the database must be accessible to more than one application. To fulfill this requirement, the Symbian OS API implements the SDP database as a server. Note that the application must connect and open a session on the server before it can access the functionality the server provides.

The `RSdp` class represents the SDP database server and allows an application to connect to it (see `bt_sdp.h`). `RSdpDatabase` represents the session on the server, and it is the more interesting of the two classes because it provides most of the functionality a client application requires.

The `CChatBtServiceAdvertiser::ConnectL()` function in the example shows how to connect to the database. The call to `CSdpSession::Connect` will establish and open a session with the server. This is then followed by a call to `Open`, to actually open the Bluetooth database.

```
void CChatBtServiceAdvertiser::ConnectL()
{
    if ( !iIsConnected )
    {
        User::LeaveIfError( iSdpSession.Connect() );
        User::LeaveIfError( iSdpDatabase.Open( iSdpSession ) );
        iIsConnected = ETrue;
    }
}
```

Both the session and the database must be closed before the client application can terminate. The session and database are closed by simply calling the `Close` function for each object.

5.1.2 Creating a service record

As previously stated, there is no explicit service record in the Symbian OS architecture. Instead, the SDP database manages a collection of service handles and their associated attributes that make up a service record.

The following line of code, taken from the function `CChatBtServiceAdvertiser::StartAdvertisingL(TInt aPort)`, illustrates how to create a service record in the database. The `iRecord` (`TSdpServRecordHandle`) is passed as a reference and will contain the service handle of the newly created service record.

```
iSdpDatabase.CreateServiceRecordL( ServiceClass(), iRecord );
```

The `CreateServiceRecordL` function provides two overloaded versions, as shown in the definition above. Both versions create service records in the database. However, the first version demonstrated previously assigns only one service class to the ID, while the second overloaded version will show a service record as being an instance of many service classes. Often it is more appropriate to advertise a service as a list of service record IDs. Each ID represents a more specific instance of a general service class. This allows service type searching. For instance, a color printer would advertise its service as being a printing device and, more specifically, a color-printing device. Both of these service class UUIDs would be represented in the list submitted to the second overloaded function.

5.1.3 Populating the service record

Although the service record exists in the SDP database, it does not have any associated attributes assigned to it. Populating the service record requires the use of the data element types described earlier.

The first element that is added to the service record is a list of protocols to which the service being advertised adheres. Creating the protocol list involves the use of the `CAttrValueDES` list introduced earlier.

5.1.4 CSdpAttrValueDES and CSdpAttrValueDES

The UML diagram in Figure 3 illustrates that both `CSdpAttrValueDES` and `CSdpAttrValueDEA` have an architecture that is slightly different from the data element types.

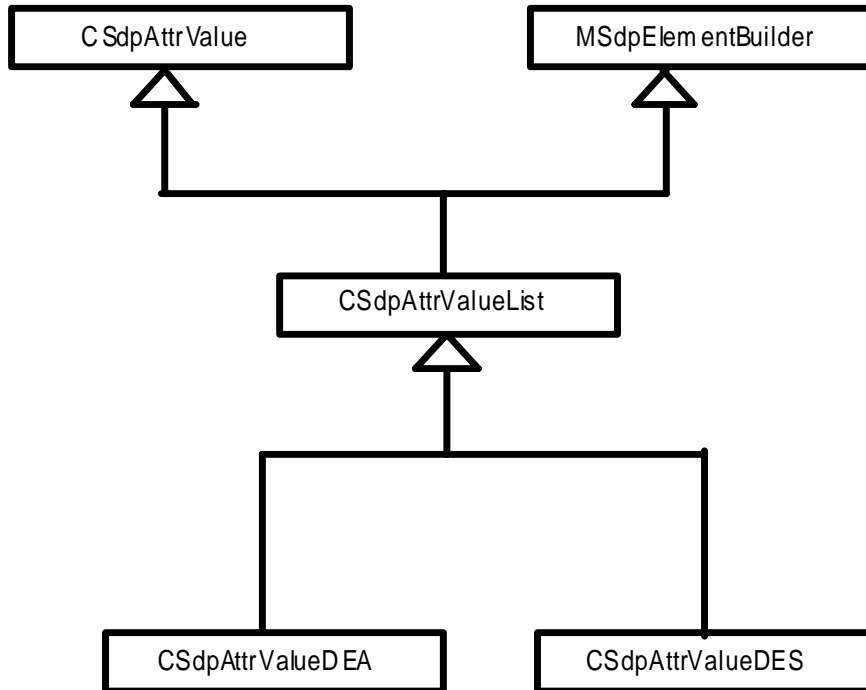


Figure 3: Architecture of CSdpAttrValueDES and CSdpAttrValueDEA

The following code explains the entire available API supplied by the CSdpAttrValueDES class:

```

class CSdpAttrValueDES : public CSdpAttrValueList
{
public:
    IMPORT_C static CSdpAttrValueDES*
        NewDESL(MSdpElementBuilder* aBuilder);
    virtual TSdpElementType Type() const;
    IMPORT_C virtual void
        AcceptVisitor1(MSdpAttributeVisitor& aVisitor);
    IMPORT_C virtual TUint DataSize() const;
    IMPORT_C void AppendValueL(CSdpAttrValue* aValue);
    IMPORT_C virtual MSdpElementBuilder*
        BuildUnknownL(TUint8 aType,
            TUint8 aSizeDesc,
            const TDesC8& aData);
    IMPORT_C virtual MSdpElementBuilder* BuildNilL();
    IMPORT_C virtual MSdpElementBuilder* BuildUinL(const
        TDesC8& aUin);
    IMPORT_C virtual MSdpElementBuilder* BuildUUIDL(const
        TUUID& aUUID);
    IMPORT_C virtual MSdpElementBuilder* BuildBooleanL(TBool
        aBool);
    IMPORT_C virtual MSdpElementBuilder* BuildStringL(const
        TDesC8& aString);
    IMPORT_C virtual MSdpElementBuilder* BuildDESL();
    IMPORT_C virtual MSdpElementBuilder* BuildDEAL();
    IMPORT_C virtual MSdpElementBuilder* StartListL();
    IMPORT_C virtual MSdpElementBuilder* EndListL();
    IMPORT_C virtual MSdpElementBuilder* BuildURLL(const
        TDesC8& aString);
    IMPORT_C virtual MSdpElementBuilder* BuildEncodedL(const
        TDesC8& aString);
    virtual TSdpElementType Type() const=0;
    virtual TUint DataSize() const=0;
    virtual TUint Uint() const;
    virtual TInt Int() const;
}
  
```

```

virtual TBool DoesIntFit() const;
virtual TInt Bool() const;
virtual const TUUID &UUID() const;
virtual const TPtrC8 Des() const;
...
};

```

Creating a data element sequence (DES) requires using the `StartListL`, `EndListL`, and `BuildxxxxL` functions. The `StartListL` function opens and returns a new DES list in the current list sequence. Using the `BuildxxxxL` function, it is possible to populate this list with actual data. When the current DES list has been correctly populated, calling the `EndListL` function will close it and return its parent list.

The following code taken from the `Chat` example shows the process of constructing a `CSdpAttrValueDES`:

```

CSdpAttrValueDES* vProtocolDescriptor =
CSdpAttrValueDES::NewDESL( NULL );
CleanupStack::PushL( vProtocolDescriptor );

BuildProtocolDescriptionL( vProtocolDescriptor, aPort );

iSdpDatabase.UpdateAttributeL( iRecord,
KSdpAttrIdProtocolDescriptorList,
*vProtocolDescriptor );

CleanupStack::PopAndDestroy( vProtocolDescriptor );

```

Once the list has been created, it must be populated with attributes. As stated before, this is done with the `StartL`, `EndListL`, and `BuildxxxxL` functions.

The following code demonstrates this process. The code is actually a C++ statement. The sample creates a “parent” list that has two child lists. The first child list contains a UUID with the value of `KL2CAP`. The second child list contains a UUID with the value `KRFCOMM`, and a second member that is an unsigned 8-bit integer with the value `KChannel`. `KChannel` has been declared with the value “1” in this example application. The use of `TBuf8` helps to explicitly designate the number of bits to use when encoding the value.

```

void CChatServiceAdvertiser::BuildProtocolDescriptionL(
CSdpAttrValueDES* aProtocolDescriptor, TInt aPort )
{
TBuf8<1> channel;
channel.Append( ( TChar )aPort );

aProtocolDescriptor
->StartListL()
->BuildDESL()
->StartListL() // Details of lowest level protocol
->BuildUUIDL( KL2CAP )
->EndListL()

->BuildDESL()
->StartListL()
->BuildUUIDL( KRFCOMM )
->BuildUIntL( channel )
->EndListL()
->EndListL();
}

```

`KL2CAP` and `KRFCOMM` are constant values that represent parts of the Bluetooth stack. Their actual value is defined in the *Bluetooth Specification*. These constant values are defined in the `bt_sock.h` header file.

Once the descriptor list has been created, it must be attached to the actual service record. This operation can be seen in the following statement:

```
iSdpDatabase.UpdateAttributeL( iRecord,
KSdpAttrIdProtocolDescriptorList,
    *vProtocolDescriptor );
```

The `CSdpDatabase` interface has already been introduced. The `KSdpAttrIdProtocolDescriptorList` constant value is part of the *Bluetooth Specification* and defines one of the universal attributes that accompany every service record. It is declared in the `btstdp.h` header file.

The `UpdateAttributeL` functions take the following structure:

```
UpdateAttributeL( <Service Record Handle to update>,
    <Attribute ID of the Service Record to update>,
    <value to assign > );
```

Note that `UpdateAttributeL` never takes ownership of any values. Therefore, it is always necessary to destroy any data element values created in the process of updating a service record. To make things easier, the overloaded versions of the `UpdateAttributeL` function will accept unsigned integers or long and short descriptors. The remaining initialization of the service record shows the use of the overloaded versions of the `UpdateAttributeL` functions:

```
// Add a name to the record
iSdpDatabase.UpdateAttributeL( iRecord,
    KSdpAttrIdBasePrimaryLanguage +
    KSdpAttrIdOffsetServiceName,
    ServiceName() );

// Add a description to the record
iSdpDatabase.UpdateAttributeL( iRecord,
    KSdpAttrIdBasePrimaryLanguage +
    KSdpAttrIdOffsetServiceDescription,
    ServiceDescription() );
```

Again, all service record attribute IDs are defined in the *Bluetooth Specification* and declared in the `btstdp.h` header file.

Note that the visibility of a Bluetooth service can be defined. The service can be such that it can only be connected to by using the exact service ID. It can be made public by adding it to a Public Browse Group list by setting an attribute to the service to enable remote devices to see it when browsing for services. Typically proprietary services (specific to a certain application) could be "hidden," as the counterpart can find and connect to them because the service ID is known. The Public Browse Group UUID is defined in `btstdp.h`. To add a service to the Public Browse Group list, use the following code:

```
TUUID browseUUID(KPublicBrowseGroupUUID)
CSdpAttrValueUUID* browseGroupAttr =
CSdpAttrValueUUID::NewUUIDL(browseUUID);
iSdpDatabase.UpdateAttributeL(iRecord,
KSdpAttrIdBrowseGroupList, *browseGroupAttr);
```

5.1.5 Removing the service advertisement

Once the service record has been added to the database, it will become available to other devices and their applications. To stop advertising a service,

the client application should delete the service record from the database. This is a relatively simple operation, performed as follows:

```
iSdpDatabase.DeleteRecordL(iRecord);
```

5.1.6 Shutting down the SDP database

When a client application has finished with its Bluetooth services, it should disconnect from the SDP database. This is done when the client application calls the `Close` function for both the Session and the database objects:

```
iSdpDatabase.Close();
iSdpSession.Close();
```

Symbian OS keeps track of all sessions that are connected to servers if they are not essential system servers. Once such a server no longer has any sessions accessing it, the operating system will shut down the server, thus conserving the limited system resources available on a handheld device. This also means that the SDP database will be shut down once there are no longer any applications with open sessions on it, even if there are other Bluetooth devices accessing it. It is therefore important to ensure that the client application keeps a session to the database open during the period it wishes to publicize its services. It is also the responsibility of the application developer to ensure that invalid SDP records are not advertised.

5.2 Changing the discoverability mode

If the client-side application in a solution takes advantage of the Limited Inquiry option for faster device discovery (discussed in detail in Section 3.2.3, "Setting the Inquiry Access Code (IAC)"), then the server-side device must also be set into Limited-discoverable mode.

Beginning with Symbian OS v8.0a (S60 2nd Edition, Feature Pack 2) the discoverability is set via the Bluetooth Publish and Subscribe API using a constant `KPropertyKeyBluetoothSetLimitedDiscoverableStatus` (from S60 3rd Edition onwards) or `KPropertyKeyBluetoothLimitedDiscoverable` (in S60 2nd Edition, Feature Packs 2 and 3). The current discoverability mode can be checked with the key `KPropertyKeyBluetoothGetLimitedDiscoverableStatus`. Usage of the Publish and Subscribe API is described in detail in Section 8.2, "Bluetooth Publish and Subscribe."

In older platforms where Bluetooth API v1 architecture is used, discoverability is set with the HCI command `static const TUint KHCIWriteDiscoverabilityIoctl`. The command can be issued by an `ioctl` call to the socket, where `KSolBtHCI` is a level parameter:

```
void Ioctl(TUint KHCIWriteDiscoverabilityIoctl, TRequestStatus&
aStatus, TDes8* aDesc=NULL, TUint aLevel=KSolBtHCI);
```

The current discoverability mode can be checked with the `static const TUint KHCIReadDiscoverabilityIoctl` command. This `ioctl` command takes no parameters.

5.3 Bluetooth registry

The Bluetooth registry stores information about found devices and their link keys. The Bluetooth device registry API has changed significantly under the Bluetooth API v2 architecture; `BTREGISTRY.DLL` has been removed, and there are new registry

subsessions that provide this functionality instead. The new APIs have been designed to remove much of the repetition that was required in UI code when performing typical UI tasks (such as finding all bonded devices) and to improve responsiveness. This is therefore an area where a more considered redesign of existing client code may be more appropriate than a straight porting operation.

Most operations on the device registry are now asynchronous, and it is suggested that most implementers will find it useful to use active objects for requests and updates to the registry in order that the user interface be kept responsive.

The classes `RBTRegServ` (registry access session, which holds an `RBTMan` session), `RBTRegistry` (opens a subsession on the BT Registry Server for remote devices), `TBTRegistrySearch` (sets search criteria on the Bluetooth Registry), and `CBTRegistryResponse` (helper class that retrieves a set of results from Bluetooth Registry as a view) defined in `btmanclient.h` provide the methods required for most registry operations.

5.3.1 Adding a device

Adding a device to the registry is not an operation that most UIs will need to implement because devices are added automatically upon pairing. The code snippets below, however, provide a useful overview of how to use the registry API.

All registry operations first need a session with the registry server:

```
RBTRegServ regServ;
User::LeaveIfError (regServ.Connect ());
```

and a subsession for the particular operation in question:

```
RBTRegistry view;
User::LeaveIfError (view.Open (regServ));
```

Now a `CBTDevice` object can be added (shown as a synchronous operation for clarity; should be done using an Active Object instead):

```
TRequestStatus stat;
view.AddDeviceL (aDevice, stat);
User::WaitForRequest (stat);
User::LeaveIfError (stat.Int ());
```

The subsession (and the session, if no longer required) should now be closed:

```
view.Close ();
regServ.Close ();
```

5.3.2 Creating a view

Most registry operations, other than adding a device, will operate on a set (a *view*) of found devices. Views can be created by searching the registry based on some criteria using the method `RBTRegistry::CreateView`. These criteria are defined in a `TBTRegistrySearch` object, for example.

```
//Assuming regServ & view setup as above
//& in CActive object [with iStatus etc]
TBTRegistrySearch searchPattern;
searchPattern.FindAll(); //will find all devices in the table
view.CreateView (searchPattern, iStatus);
SetActive ();
```

If this operation completes successfully, the subsession (a view) now points at a rowset matching the query defined by the `TBTRegistrySearch`. Other methods on `TBTRegistrySearch` to set selection criteria are listed in Table 4.

Function	Meaning
<code>Reset()</code>	Clear out search settings.
<code>FindAll()</code>	Request search for all devices.
<code>FindAddress(const TBTDevAddr& aAddress)</code> <code>aAddress</code> - address of device to search for	Request search for given device.
<code>FindBonded()</code>	Request search for bonded devices.
<code>FindTrusted()</code>	Request search for trusted devices.
<code>FindCoD(const TBTDeviceClass& aClass, TBTDeviceClassSearch aPref)</code> <code>aClass</code> - the device class to search for <code>aPref</code> - describes the type of device search to perform (see <code>TBTDeviceClassSearch</code> in source docs)	Request search by the class of device.
<code>FindBluetoothName(const TDesC8& aName)</code> <code>aName</code> - device name to search for	Request search for devices with given name.
<code>FindFriendlyName(const TDesC& aName)</code> <code>aName</code> - friendly name to search for	Request search for devices with given friendly name.
<code>FindCurrentProcessOwned()</code>	Request search for devices added by this process.

Table 4: Methods on `TBTRegistrySearch` to set the selection criteria

Once a view has been created on the registry, applications will either need to retrieve the `CBTDevice` objects within that view or perform an operation on all devices in the view.

5.3.3 Retrieving results

The set of devices resulting from a registry search (that is, in the current view) can be retrieved by using the utility class `CBTRegistryResponse`, to acquire an `RBTDeviceArray` of `CBTDevice` objects. Note that `RBTDeviceArray` is a type definition for `RPointerArray<CBTDevice>`, and clients take ownership of the `CBTDevice` pointers returned in this array. See the system documentation for `RPointerArray` for more details.

```

CMyExampleAO::RetrieveResultsL()
{
    //iRegistryResponse a CBTRegistryResponse* ,

```

```

//iRegistry an RBTRegistry&
iRegistryResponse = CBTRegistryResponse::NewL(iRegistry);
iRegistryResponse->Start(iStatus);
SetActive();
}

CMyExampleAO::RunL()
{
//Called back when results have been got
RBTDeviceArray results = iRegistryResponse->Results();
//do something with results here
results.ResetAndDestroy(); //we owned the CBTDevice* array
}

```

5.3.4 Deleting and unpairing devices

Some operations act on the entire view rather than on individual devices, namely deletion and unpairing. Deletion of the devices in a view is accomplished thus:

```

//view is an RBTRegistry which has
//had CreateView called as in 5.3.2 above
view.DeleteAllInView(iStatus);

```

and likewise unpairing:

```

//view is an RBTRegistry which has had
//CreateView called as in 5.3.2 above
view.UnpairAllInView(iStatus);

```

Note that in future releases of Symbian OS these operations may require the client process to have greater capability than, say, retrieving devices from the registry.

It should be noted then, that to delete a single `CBTDevice` object from the registry it is necessary to carry out a two-phase operation: create a view containing that (single) object, and delete all objects in the view.

This is illustrated below (synchronous for clarity, but better to use Active Object with two states):

```

//view as in section 5.3.1 above.
TBTRegistrySearch search;
//the bd_addr of the device to be deleted
search.FindAddress(aAddress);
TRequestStatus stat;
view.CreateView(search, stat);
User::WaitForRequest(stat);
User::LeaveIfError(stat.Int());
view.DeleteAllInView(stat);
User::WaitForRequest(stat);
User::LeaveIfError(stat.Int());
//device has now been deleted

```

5.3.5 Modifying devices

The API for modifying device entries in the Registry consists of `RBTRegistry::ModifyDevice` and `RBTRegistry::ModifyBluetoothDeviceNameL`.

Applications may wish to assign a short, friendly name to a device. This name may be modified by use of the method

```
RBTRegistry::ModifyFriendlyDeviceNameL.
```

```
//registry is an open
```

```
//RBTRegistry object as in section 5.3.1 above.
//aDevAddr is a TBTDevAddr, aNewName a TDesC&:
registry.ModifyFriendlyDeviceNameL(aDevAddr, aNewName, iStatus);
SetActive();
```

5.3.6 Communication port settings

Under the Bluetooth API v1 architecture, the functions `CBTRegistry::SetDefaultCommPort` and `CBTRegistry::GetDefaultCommPort` were used to configure the Bluetooth CSY. Any code calling these methods should be ported to use the new `RBTCommPortSettings` class — this is another sub-session to the registry and can be similarly initialized. See the source documentation for more information on this API. The example below shows synchronous usage for illustrative purposes:

```
RBTRegServ registry;
User::LeaveIfError(registry.Connect());
CleanupClosePushL(registry);
RBTCommPortSettings setter;
User::LeaveIfError(setter.Open(registry));
CleanupClosePushL(setter);
TBTCommPortSettings settings;
settings.SetPort(0);
settings.SetBTAddr(TBTDevAddr(MAKE_TINT64(0x1122, 0x334427)));
settings.SetUUID(TUUID(0x00001101));
TRequestStatus status;
setter.Update(settings, status);
User::WaitForRequest(status);
User::LeaveIfError(status.Int());
setter.Delete(settings, status);
User::WaitForRequest(status);
User::LeaveIfError(status.Int());
User::LeaveIfError(setter.Get(settings));
CleanupStack::PopAndDestroy(2); // registry, setter
```

5.3.7 Overview of the older architecture

In the older architecture, `btregistry.h` (and `btdefcommport.h`) formed the Bluetooth Registry API but is now removed. Class `CBtRegistry` was the key class in Bluetooth Registry v1 architecture. Its main functionality is provided by the `Add`, `Delete`, and `Retrieve` functions, which take `CBtDevice` parameters.

6. Communication using sockets

The RFCOMM protocol emulates RS-232 serial ports over the L2CAP protocol, and thus it is quite useful for many occasions, especially with connections to legacy applications on the PC, but it can also be used easily in communication between devices. The data is sent as a stream over RFCOMM.

L2CAP is layered over the Baseband Protocol and resides in the data link layer.

Access to both the RFCOMM and L2CAP transmission protocols is provided by the Symbian OS socket architecture. RFCOMM uses sockets of the stream socket (`KSocketStream`) type. L2CAP uses sockets of the sequenced packet (`KSocketSeqPacket`) type.

The Symbian OS API provides a set of Bluetooth sockets that allow applications to communicate over the Bluetooth medium. The Bluetooth socket component extends the existing socket (`RSocket`) architecture in the operating system. In addition, the Bluetooth API v2 architecture also provides a new `CBluetoothSocket` wrapper class for easier access to all Bluetooth socket functions.

Once the device and service have been established, you can connect to the remote service and start using it. Connecting to the device is done through the `Connect()` function of the generic Symbian OS socket interface (`RSocket` or `CBluetoothSocket`). Bluetooth sockets can be opened using the L2CAP protocol or RFCOMM. Data can be read and written using the socket in whatever format the target service expects (AT commands, text, HTTP, PPP, etc.). For an L2CAP Bluetooth socket, the "port" to connect to is the Protocol/Service Multiplexer (PSM); for RFCOMM, the port is the server channel. Where these values are not known, they can be read from the service attribute `ProtocolDescriptorList`.

Describing the entire socket architecture is beyond the scope of this document. Instead, this document strives to introduce the fundamental components of the Bluetooth sockets architecture that are required for data transmission.

The Symbian OS API provides a specific Bluetooth socket address object, which inherits and extends the existing socket type. This `TBTSocketAddr` class is defined in the `bt_sock.h` header file. Aside from the constructors, it defines two new functions that allow a client application to get and set the socket address using the 48-bit Bluetooth address structure `TBTDevAddr`.

6.1 CBluetoothSocket

Beginning in Symbian OS v8.0a, a new `CBluetoothSocket` wrapper class has been provided for easy access to all Bluetooth socket operations (data transfer and configuration of the link).

The class contains an `RSocket` object and supports the `RSocket` methods as forwarding calls to that object, but it is created with a reference to an `MBluetoothSocketNotifier`, which handles all asynchronous completions of `RSocket` operations in its virtual methods.

Depending on the Bluetooth hardware support, low power and master/slave-switch features can also be accessed using this class.

It is possible to continue to use the `RSocket` API as before and other APIs (for example, `RBTPhysicalLinks`) to configure the additional functionality, however it is recommended that, where appropriate, application code be migrated to use `CBluetoothSocket` objects in place of `RSocket` objects.

In Symbian OS v9.1, new static factory methods have been added:

```
CBluetoothSocket* NewL(MBluetoothSocketNotifier& aNotifier,
                      RSocketServ& aServer, RSocket& aSocket);
CBluetoothSocket* NewLC(MBluetoothSocketNotifier& aNotifier,
                       RSocketServ& aServer, RSocket& aSocket);
```

There are also new versions of `Connect()` and `Listen()` functions. Additionally, the following has been added:

```
TInt SetOption(TUint aOptionName, TUint aOptionLevel, const TDesC8&
aOption)
```

Functions `TInt Ioctl(TUint aCommand, TDes8* aDesc=NULL, TUint aLevel=KLevelUnspecified)` and `TInt SetOpt(TUint aOptionName, TUint aOptionLevel, const TDesC8& aOption=TPtrC8(NULL, 0))` have been deprecated.

6.2 Device name length

Previously, the Bluetooth device name was stored in the `CBTDevice` class as a Unicode descriptor. This is now stored as an 8-bit (UTF-8 encoded) descriptor. A helper class (`BTDeviceNameConverter`) is provided for ease of conversion between the formats. It is recommended that this class be used so that source compatibility (SC) may be better preserved in the future. An example of its usage:

```
//iDevice is a CBTDevice pointer
#ifdef __BLUETOOTH_API_V2__ // macro not declared until S60 3rd Ed
    iDevice->SetDeviceNameL(BTDeviceNameConverter::ToUTF8L(name));
#else
    iDevice->SetDeviceName( name );
#endif

//getting the device name
#ifdef __BLUETOOTH_API_V2__ // macro not declared until S60 3rd Ed
    TBTDeviceName name = BTDeviceNameConverter::ToUnicodeL(
        iDevice->DeviceName()
    );
#else
    TDesC name = iDevice->DeviceName();
#endif
```

`TBTDeviceName` is defined as

```
typedef TBuf<KMaxBCBluetoothNameLen> TBTDeviceName;
```

These name length changes particularly affect the Bluetooth UI notifiers and UI control panel-type applications.

6.3 Master, slave, and role switching

As discussed earlier, in piconet there can be only one master and seven slaves. The master has to establish the connections to the slaves. `CBluetoothSocket` contains the functions for role switching:

```
AllowRoleSwitch();
RequestMasterRole();
RequestSlaveRole();
```

If the devices support role switching and there is a possibility for scatternet, then connection establishment and roles will be much more flexible.



Note: In practice, it is recommended to limit the number of connected device to four to ensure better throughput.

6.4 Options and protocol strings

The Bluetooth protocol provides a set of options that can be retrieved by using the `RSocket::GetOpt` (or `CBluetoothSocket::GetOpt`) function. The set of constants required to obtain these options is defined in the `bt_sock.h` header file. Protocols in the sockets architecture can also be identified either with operating system-defined constants or by using a string. The RFCOMM protocol can be identified with the `RFCOMM` string and L2CAP with the `L2CAP` string. The choice between RFCOMM and L2CAP is up to the developer. The examples in this document mostly use RFCOMM.

6.5 Listening sockets

Only minor changes in listening methods have been introduced in Symbian OS v8.0a. It is recommended to use the `CBluetoothSocket` wrapper class instead of `RSocket`.

Note that the method `RSocket::SetLocalPort(port number)` has been deprecated for security reasons.

```
//SockServ is an RSockServ&, aSec a TBTServiceSecurity
CBluetoothSocket listener;
TProtocolDesc pDesc;
// KRFCOMMDesC("RFCOMM") or KL2CAPDesC("L2CAP"), see bt_sock.h
User::LeaveIfError(sockServ.FindProtocol(KRFCOMMDesC, pDesc));
listener.Open(sockServ, pDesc.iAddrFamily, pDesc.iSockType,
             pDesc.iProtocol);
TRfcommSockAddr addr;
```

To get the first available port:

```
Addr.SetPort(KRFCOMMPassiveAutoBind);
```

To set the security if needed (see Section 8.1, "Bluetooth Security Manager," for details):

```
Addr.SetSecurity(aSec);
User::LeaveIfError(listener.Bind(addr));
```

This method can be used to find out the number of the port that has been allocated to the application:

```
TUint port = listener.LocalPort();
StartAdvertising(port);
```

Listening can be started by calling:

```
User::LeaveIfError(listener.Listen(1));
```

6.5.1 Listening sockets in Bluetooth API v1 architecture

In S60 1st and 2nd Editions there is a `CMessageServer::StartL` function in the `\series60Ex\BTPointToPoint\src\MessageServer.cpp` file that performs all the work of starting up the RFCOMM service. Initially, it starts the Bluetooth sockets protocol and initializes an RFCOMM port. First, check to see if Bluetooth is on (see Section 8.4, "Checking Bluetooth power mode").

```
void CMessageServer::StartL()
{
```

```
//...
```

To open a server socket, the device must first connect to the socket server with `RSocketServ`; then open a connection to this server with an `RSocket` object. When the client application calls the `Open` function on the `RSocket` object, it must specify the protocol it wishes to use. In this example, `CMessageServer` specifies the protocol using the string literal `KServerTransportName`, which equates to `RFCOMM`.

```
User::LeaveIfError(iSocketServer.Connect());
```

```
User::LeaveIfError(iListeningSocket.Open(iSocketServer,
    KServerTransportName));
```

The next task is to query the protocol for an available server channel. This is accomplished by calling the `GetOpt` function on the `RSocket` object. The client application specifies the type of option request in the first parameter, and the protocol in the second. When the function completes, the third parameter contains an integer representing the available server channel.

```
// Get a channel to listen on - same as the socket's port
// number
TInt channel;
User::LeaveIfError(
    iListeningSocket.GetOpt(KRFCOMMGetAvailableServerChannel,
        KSolBtRFCOMM, channel));
```

The client application then composes the `TBTSockAddr` object specifying the port it has just obtained from the protocol options.

```
TBTSockAddr listeningAddress;
listeningAddress.SetPort(channel);
iLog.LogL(_L("Get port= "), channel);
```

`CMessageServer` binds the newly created `RSocket` to the port it has just specified. It now sets the socket up to listen for any incoming data.

```
User::LeaveIfError(iListeningSocket.Bind(listeningAddress));
User::LeaveIfError(iListeningSocket.Listen(KListeningQueueSize));
//...
```

`CMessageServer` has been implemented using an Active Object. The `RSocket::Accept` function informs the operating system that it should call the `CMessageServer::RunL` function when incoming data has been received.

```
iState = EConnecting;
iListeningSocket.Accept(iAcceptedSocket, iStatus);
SetActive();
iLog.LogL(_L("Accept next Connection"));
```

Once the RFCOMM socket has been opened successfully, the client application configures the security manager. This process is outlined in Chapter 8, "Security and configurations."

```
SetSecurityOnChannel(EFalse, EFalse, ETrue, channel);
```

The client application advertises its new RFCOMM service only after the port and its security have been set up. This process is outlined in Chapter 5, "Discoverability and advertising."

```
iAdvertiser->StartAdvertisingL(channel);
iAdvertiser->UpdateAvailabilityL(ETrue);
}
```

Once a connection with a client has been accepted, `CMessageServer` must ensure that the RFCOMM entry in the SDP database is marked as being used. This ensures that a second client will not attempt to connect to the same port. Marking the entry is simply a matter of updating an attribute value in the service record.

6.6 Transmitting socket

Setting up a transmitting socket is performed in a manner similar to the receiving example discussed in Section 6.5, "Listening sockets."

In the Chat example [3], the `CChatBt` class sets up the RFCOMM socket.

The `CChatBt` class connects and opens a socket on the RFCOMM protocol in an identical manner.

```
iSocketServer.Connect(); //in ConstructL

//in ConnectToServerL function:
iSocket.Open( iSocketServer, KStrRFCOMM );
//for L2CAP the following could be used:
//iSocket.Open(iSocketServer, KBTAddrFamily, KSockSeqPacket,
                KL2CAP));
```

The client application now composes the address of the device it wishes to connect to. It is assumed that the Bluetooth device discovery process has identified the address of the device offering the RFCOMM service. In addition, this process should have identified the Bluetooth channel on the receiving device to which the client application should connect. The discovery process is covered in more detail in Chapter 4, "Service discovery."

```
//in ConnectToServerL:
TBTSockAddr address;
address.SetBTAddr( iServiceSearcher->BTDevAddr() );
address.SetPort( iServiceSearcher->Port() );
```

Once the address has been composed, it is used to open the connection to the remote device:

```
iSocket.Connect(address, iStatus);
//...
SetActive();
}
```

6.7 Sending data

Sending data over an RFCOMM connection is simply a matter of writing to the connected `RSocket` or `CBluetoothSocket`. The `CChatBt::SendMessageL` function performs this task.

```
CChatBt::SendMessageL( TDes& aText )
{
...
iSocket.Write(*iMessage, iStatus);
SetActive();
}
```

6.8 Receiving data

Once a connection with a client device has been established, the receiving device need only call the `RecvOneOrMore` function on the socket the client has connected to. This

function will call the `CChatBt::RunL` function when data has been received. In the `Chat` example, this is implemented in the `RequestData` function.

```
void CChatBt::RequestData()
{
    if ( iActiveSocket )
    {
        iActiveSocket->RecvOneOrMore( iBuffer, 0, iStatus, iLen );
    }
    SetActive();
}
```

In cases where `CBluetoothSocket` is used, then the `CBluetoothSocket::RecvOneOrMore(TDes8 &aDesc, TUint flags, TSockXfrLength &aLen)` is called. The observer, which implements the `MBluetoothSocketNotifier` interface, is notified when the action completes.

6.9 Shutting down the service

Shutting down a Bluetooth socket is simply a case of calling `close` on the `RSocket` or `CBluetoothSocket` object. Before this, the client application must ensure that all services have been removed from the SDP database. Note that for both the RFCOMM and L2CAP protocols, no data can be sent or received in the `Shutdown()` (and `Connect()`) calls.

7. Communication using OBEX

OBEX provides a method for transferring objects or chunks of data from one device to another. These chunks are typically files or other blocks of binary data. OBEX runs on top of a number of different transport mediums, including IrDA and Bluetooth RFCOMM.

Note that Bluetooth sockets are a more preferable choice when there is a need for real communication between devices. OBEX is more suitable for single-shot operations like transferring a file.

This chapter provides developers with the necessary information they will need to implement a simple OBEX transmission using Bluetooth, and, specifically, RFCOMM as the transport medium.

The OBEX communications API is present in Symbian OS. It is implemented with these five main classes:

- `TObexBluetoothProtocolInfo`
- `TObexConnectInfo`
- `MObexServerNotify`
- `CObexClient`
- `CObexServer`

These classes are defined in the `obexXXX.h` header files, which are included in the `obex.h` file. They will be covered in more detail in the sections that follow.

Note that there is also an S60 API that provides an alternative, limited way of using OBEX in the application when only sending of a file to another device is needed; see Section 7.6, "Easy way of using OBEX with Send UI."

Forum Nokia provides an OBEX example [4] that consists of a single application that can act as a server or as a client. Since Bluetooth is used for data transmission there is a need to have a Bluetooth service in the server side and a Bluetooth device and service discovery in the client side. The server advertises a Bluetooth service, which the client discovers and connects to. After connecting, the client can send OBEX objects to the server. The example implements file sending using the `CObexFileObject` class.

7.1 Defining Bluetooth protocol

This class provides a way for a client application to specify the underlying Bluetooth protocol it wishes OBEX to use. The class is based on the `TObexProtocolInfo` base class. Its full public interface is as follows (the code excerpt does not actually represent the real implementation of the class):

```
class TObexBluetoothProtocolInfo : public TObexProtocolInfo
{
public:
    TRfcommSockAddr iAddr;
    TBuf<60> iTransport; // from TObexProtocolInfo
};
```

When using the `TRfcommSockAddr` object to set up a Bluetooth RFCOMM OBEX session, `iAddr` represents the channel or port number to connect to on the remote device. The `iTransport` field allows the client application to specify a string identifying the protocol to use.

In the OBEX example, `TObexBluetoothProtocolInfo` is used when the server is started:

```
void CObjectExchangeServer::InitialiseServerL()
{
    if ( iObexServer )
    {
        ASSERT( IsConnected() ); // server already running
        return;
    }

    TInt channel (
        SetSecurityWithChannelL( EFalse, EFalse, ETrue, EFalse ) );

    // start the OBEX server
    TObexBluetoothProtocolInfo obexProtocolInfo;
    obexProtocolInfo.iTransport.Copy( KServerTransportName );
    obexProtocolInfo.iAddr.SetPort( channel );

    iObexServer = CObexServer::NewL( obexProtocolInfo );
    iObexServer->Start( this );

    // advertise this service
    iAdvertiser->StartAdvertisingL( channel );
    iAdvertiser->UpdateAvailabilityL( ETrue );
    //...
}
```

`TObexBluetoothProtocolInfo` is also used when connection to the server is established:

```
void CObjectExchangeClient::ConnectToServerL()
{
    TObexBluetoothProtocolInfo protocolInfo;

    protocolInfo.iTransport.Copy( KServerTransportName ); // "RFCOMM"
    protocolInfo.iAddr.SetBTAddr( iServiceSearcher->BTDevAddr() );
    protocolInfo.iAddr.SetPort( iServiceSearcher->Port() );
    //...
    iClient = CObexClient::NewL( protocolInfo );

    iClient->Connect( iStatus );
    SetActive();
}
```

Once a connection has been established, the `TObexConnectInfo` class will supply version information about the OBEX version that is used by the connecting device.

7.2 OBEX objects

All data transmitted over OBEX is wrapped up in a containing object before it is sent. There are three main types of OBEX data wrapper classes. `CObexBaseObject` provides the base class for all of these data wrappers. `CObexBaseObject` is a virtual class that cannot be instantiated. The three concrete classes are covered in the following list:

- `CObexFileObject` is designed to be used when sending files over OBEX.
- `CObexBufObject` is meant to be used when sending a chunk of memory over OBEX.
- `CObexNullObject` provides a means of sending a blank object.

All three classes derive from the same base class, and all have similar APIs. In the interest of brevity, the individual APIs are not presented in this document, but they can be found in the `obex.h` header file.

The OBEX example [4] uses `COBexFileObject` to send a file from the client to the server:

```
void CObjectExchangeClient::SendObjectL(TFileName& aName)
{
    if ( iState != EWaitingToSend )
    {
        User::Leave( KErrDisconnected );
    }
    else if ( IsActive() )
    {
        User::Leave( KErrInUse );
    }
    delete iCurrObject;
    iCurrObject=NULL;

    //Set the file when creating the object
    iCurrObject = COBexFileObject::NewL(aName);
    TParsePtr parsePtr (aName);
    TPtrC ptr = parsePtr.NameAndExt();

    //Set the name to be the file name without the path
    iCurrObject->SetNameL( ptr );

    //Send the object
    iClient->Put( *iCurrObject, iStatus );
    SetActive();
}
```

In the OBEX example the user is asked to select a file to be sent (note that this does not apply to S60 1st Edition, since memory and file selection headers `caKnmemoryselectiondialog.h` and `caKnfileselectiondialog.h` were not public in the S60 1st Edition [1.2] SDK).

7.3 OBEX server notification mechanism

`MOBexServerNotify` is a Symbian OS observer class. It is used by the operating system to inform a client application of OBEX communication events. Typically, an OBEX server would implement this class.

The `MOBexServerNotify` observer class defines all of its functions as private. This implies that the `MOBexServerNotify` class can only be used as an observer with the `COBexServer` class. In the OBEX example [4], `CObjectExchangeServer` implements the `MOBexServerNotify` interface.

The observer class defines a number of observer functions. The following list provides a brief explanation of the functions that might interest a Bluetooth developer. The most important functionality in the OBEX example server is implemented in these functions:

- **ErrorIndication** – This function is called when an OBEX error occurs. It receives an integer value that will contain one of the standard Symbian error codes. In the examples server class `CObjectExchangeServer` the error is just shown to the user.
- **TransportUpIndication** – This function is called by the operating system when a client makes a connection to the server. It does not indicate that an OBEX session has begun, merely that a connection has been made in the underlying protocol, in this case, Bluetooth. In `CObjectExchangeServer`, this event is shown in the application view.
- **TransportDownIndication** – The operating system calls this function when the underlying protocol connection has been lost, not when the OBEX session has completed successfully. In `CObjectExchangeServer`, the

OBEX object (`iObexBufObject`) is set to use a data buffer instead of the file to prevent file recreation.

- `ObexConnectIndication` – The operating system calls this function when the OBEX connection has actually been set up; the `aRemoteInfo` parameter provides version information about the connecting device.
- `ObexDisconnectIndication` – This function is called when the OBEX connection has successfully disconnected. It is not called if the connection is unexpectedly lost. In such an event the client application can expect that the `ErrorIndication` function will be called.
- `PutRequestIndication` – When the connection has been established, the receiving device's operating system calls this function. This function allows the client application to reject the connection request. This is done by returning NULL from the function. Alternatively, the client application should return a pointer to a concrete instance of the `COBexBaseObject` class; this accepts the incoming request. The returned object is populated by the incoming data. In the OBEX example, the server creates a `COBexBufObject` and sets its data buffer to a file by calling `SetDataBufL(*iTempFile)`. This way the received file will be written to the specified location.
- `PutPacketIndications` – This function is called once for every packet received from the remote device. It allows a client application to provide details of the transfer to the user. Since the function will be called frequently, it is critical that its implementation is as fast as possible.
- `PutCompleteIndication` – When this function is called, the application can be assured that all the OBEX packets have been received from the remote device. If the connection breaks or an error occurs, this function is not called, but the `ErrorIndication` function is called instead. In the OBEX example, the file transfer has finished when this function is called. `COBjectExchangeServer` reads the file name (which is set to the OBEX object name) and renames the temporary file according to the new name.
- `GetRequestIndication` – When a remote device sends a request, the server application receives a call to the `GetRequestIndication` function. This allows the application to accept the request and return a pointer to the concrete instance of the `COBexBaseObject` class. This object is then transmitted to the remote device. The application can deny the request by returning a NULL value. This is the way `COBjectExchangeServer` works, since it doesn't support Get functionality.
- `GetPacketIndication` – Once the transmission of the OBEX object has begun, this function is called once for every packet sent.
- `GetCompleteIndication` – This function is called when the transmission has been completed. It allows the client application to perform any clean-up actions that may be required.
- `AbortIndication` – This function is called when the client device has aborted the transmission in a controlled manner.

7.4 OBEX client

`COBexClient` provides a client application with the necessary functionality to transmit to, and request from, an OBEX server.

When a client application constructs `COBexClient`, it must specify the address of the server device and the protocol to use to connect to it. As previously stated for Bluetooth,

this information is encapsulated in a `TObexBluetoothProtocolInfo` object. Once a client has an instance of the `CObexClient` class, it can then connect to the remote OBEX device and either request or send an OBEX object. These operations are performed by the `Connect`, `Get`, and `Put` functions. All three functions are asynchronous. Once a transmission is in progress, a client application can terminate it by calling the `Abort` function.

The creation of `CObexClient` in the OBEX example [4] is demonstrated in Section 7.1, "Defining Bluetooth protocol."

7.5 OBEX server

`CObexServer` allows a client application to offer OBEX services to other devices. Like `CObexClient`, when constructing `CObexServer`, the client application must supply `TObexProtocolInfo`. This functionality for the OBEX example [4] is shown in Section 7.1, "Defining Bluetooth protocol." The `TObexProtocolInfo` class is required to provide listening information to the server. The required information includes the protocol and the port, or, in Bluetooth terms, the channel to listen to. Both of these are encapsulated in the `TObexBluetoothProtocolInfo` class.

Once the server object has been created, the client must start it. This is done using the `Start` function. When this function is called, the client must supply an instance of the observer class `MObexServerNotify`. As stated previously, the operating system, via `CObexServer`, will inform the observer of any relevant events.

Calling the `Stop` function stops the server. Server status can be checked by calling the `IsStarted` and `CurrentOperation` functions.

7.6 Easy way of using OBEX with Send UI

Send UI is a convenient messaging API that hides the bearer details from the developer. It can be easily used for sending short message service (SMS), multimedia messaging service (MMS), and e-mail, and even for data transfer over Bluetooth or infrared.

The basic procedure in using Send UI is to add the Send command to the Options menu. The Send command bearer list (shown as a cascade menu in S60 1st and 2nd Editions) is populated with the possible bearers (like Bluetooth). The bearers that appear in the list may differ according to developer settings. If the developer wishes to use attachments, then SMS is not shown in the list. Of course the bearers may differ between different mobile phone models. Also the mobile device settings may affect the bearer visibility. For example, if there's no mailbox defined, then e-mail is not an option.

After the bearer has been selected, the sending may begin. If the bearer is Bluetooth, the Bluetooth Device Selection Notifier Plug-in is called and the list of remote devices is displayed after the user has selected the menu item. The selected file will then be sent via Bluetooth Object Push Profile (using OBEX protocol) to the device the user selects from the list.

In S60 3rd Edition, the Send UI API, provided by the S60 platform, has been cleaned up (the new API is less complex and easier to use). The UI specification of the Send menu has been updated to use the list query instead of the submenu. The `CSendAppUi` class has been removed, and `CSendUi` is introduced in the `Sendui.h` header file instead. All of the Message Type Module (MTM) values are defined in `SenduiMtmUids.h` (`KsenduiMtmBtUidValue`, the message type value for Bluetooth, is `0x10009ED5`). Don't confuse `CSendAs` (introduced in `sendas.h`) and its S60 3rd Edition version `RSendAs` (in `rsendas.h`) with `CSendAppUi` and `CSendUi`; they cannot be used with Bluetooth.

7.6.1 Send UI API usage example

The Bluetooth OBEX example [4] also demonstrates usage of the Send UI API (both `CSendUi` and the former equivalent `CSendAppUi`). Basic use of the Send UI API involves the following steps:

1. Introduce a `CSendUi` object (it is recommended to declare `CSendUi` as a member variable and initialize it in the constructor).
2. Add the Send Menu item.
3. Display the Send list query (or cascade menu in S60 1st and 2nd Editions) and create a message to be sent.

The Bluetooth OBEX example [4] shows how to use the Send UI API. The changes in S60 3rd Edition can be taken into consideration by using macros, as shown in the next code snippet:

```
#ifdef __SERIES60_3X__
iSendUi = CSendUi::NewL();
#else
iSendUi = CSendAppUi::NewL(ESendViaSendUi);
#endif
```

The construction of the objects is slightly different since `CSendAppUi` requires the Send menu item command value to be given as an argument. This is required because the sending functions in `CSendAppUi` receive the command ID reflecting the bearer that the user has selected. By knowing the Send menu item command value it is possible to calculate which bearer was selected by counting the offset.

The Send UI API is meant to be used by selecting the Send command from the application's Options menu. In S60 3rd Edition, the `CSendUi` class has an `AddSendMenuItemL` function that adds the menu item. The desired bearer options (stating that should Bluetooth, IR, SMS, MMS, etc., appear in the bearer list) are also given by passing `TSendingCapabilities` as an argument.

```
void CSendUi::AddSendMenuItemL ( CEikMenuPane & aMenuPane,
    TInt aIndex,
    TInt aCommandId,
    TSendingCapabilities aRequiredCapabilities =
    KCapabilitiesForAllServices
)
```

`aMenuPane` is a menu pane to which the Send menu item has been added. `aIndex` is the place of the Send in the menu. `aRequiredCapabilities` are the capabilities required by the MTMs to be displayed.

Adding the Send menu item can be done in the client application's `DynInitMenuPaneL()` function. The following code snippet, taken from the OBEX example, demonstrates how to add the Send menu item when the client application is using the Send UI API to send content as attachments. If there are no message types available that match to the `TSendingCapabilities`, then the Send menu item is not added to the menu pane.

The following code is taken from the Bluetooth OBEX example [4].

```
void CBTObjectExchangeAppUi::DynInitMenuPaneL( TInt
aResourceId, CEikMenuPane* aMenuPane )
{
    if ( aResourceId == R_BTOBJECTEXCHANGE_MENU )
    {
        //...
    }
}
```

```

// Display the Send menu item (if there are bearers)

// Show only the bearers which support attachments
TSendingCapabilities capabilities(0,0,
TSendingCapabilities::ESupportsAttachments );

#ifdef __SERIES60_3X__
//In S60 3rd Edition the menu item is not created in the rss
TInt position = 0;
//so find a desired command
aMenuPane->ItemAndPos(
    EBTObjectExchangeClearList, position );
//and add the Send menu item next to it
iSendUi->AddSendMenuItemL(*aMenuPane, position-1,
    ESendViaSendUi, capabilities );
#else
//In S60 1st and 2nd Edition we have the Send menu item
// with its cascade menu defined in the rss file
TInt position = 0;
aMenuPane->ItemAndPos( ESendViaSendUi, position );
iSendUi->DisplaySendMenuItemL (*aMenuPane, position,
    capabilities );
aMenuPane->DeleteMenuItem(ESendViaSendUi);
#endif
}
#ifdef __SERIES60_3X__ //this is not needed in 3rd ed.
// Show the SendUi cascade menu
else if ( aResourceId == R_BTOBEX_SENDUI_MENU)
{
    iSendUi->DisplaySendCascadeMenuL(*aMenuPane, NULL);
}
#endif
}
}

```

The use of `CSendUi` in S60 3rd Edition requires less work from the developer than the previously used `CSendAppUi`. For `CSendUi` the menu item does not need to be introduced in the resource file (`.rss`), whereas for `CSendAppUi` the menu item and its cascade menu have to be introduced.

After the menu item is visible and the user selects the menu command, the command handling is done in `HandleCommandL`:

```

void CBTOBExchangeAppUi::HandleCommandL( TInt aCommand )
{
    //...
    switch ( aCommand )
    {
        //...
        case ESendViaSendUi:
            //These are the commands from Send cascade menu
            //(actually needed only in 1st and 2nd Edition)
            case ESendViaSendUi1: //FALLTHROUGH
            case ESendViaSendUi2: //FALLTHROUGH
            case ESendViaSendUi3: //FALLTHROUGH
            case ESendViaSendUi4: //FALLTHROUGH
            case ESendViaSendUi5: //FALLTHROUGH
            case ESendViaSendUi6: //FALLTHROUGH
            case ESendViaSendUi7: //FALLTHROUGH
            case ESendViaSendUi8: //FALLTHROUGH
            case ESendViaSendUi9:
            {
                //With all the commands we send a file.
                //Pass the command to the function. It's required
                //by CSendAppUi sending function
            }
    }
}

```

```

        SendFileViaSendUiL(aCommand);
        break;
    }
    //...
}

```

Because of the changes in the Send UI API, the `ESendViaSendUi` command is the only one that is required to be handled in S60 3rd Edition. S60 1st and 2nd Editions use a cascade menu (submenu), so there are more commands to handle — in fact, as many as there are bearers available. Because it is difficult to know how many bearers might exist, it is recommended to add a sufficient number of commands to handle the different bearers. The bearer command values start from the Send menu item command value and increase in increments of one.

In the Bluetooth OBEX example [4], after the user has selected the Send menu command, the `SendFileViaSendUiL` function is called. The function is not a part of the OBEX API; it is simply a helper function in the example. With `CSendAppUi`, the command ID is needed in the sending because it describes the selected bearer. Note that there is a fundamental difference between `CSendUi` and `CSendAppUi`. With `CSendAppUi`, the bearer is already selected by the user (from the cascade menu) as the `HandleCommandL` function is executed. With `CSendUi`, the selection is done afterwards from the bearer list.

Displaying the Send list query and creating a message can be done together or separately. The `CMessageData` class is used to encapsulate the message data, for example, recipient addresses, message subject and body text, and attachments. Unnecessary message types can be filtered away from the Send list query by defining `TSendingCapabilities` and/or an array of filtered message type IDs.

```

void CBTObjectExchangeAppUi::SendFileViaSendUiL(TInt aCommand)
{
    TFileName path;

    if (AskFileL(path) )
    {
        TSendingCapabilities capabs( 0, 1024,
            TSendingCapabilities::ESupportsAttachments );

#ifdef __SERIES60_3X__

        RFile handle;

        //...
        CMessageData* messageData = CMessageData::NewL();
        CleanupStack::PushL(messageData);
        //there's an open file handle to `path`
        messageData->AppendAttachmentHandleL(handle);

        //The actual sending is done in this function.
        //First the bearer list query is shown.
        iSendUi->ShowQueryAndSendL(messageData, capabs);

        CleanupStack::PopAndDestroy(messageData);
        CleanupStack::PopAndDestroy(&handle);
        //...

    #else //1st and 2nd Edition
        const TInt KGranularity = 2;
        CDesC16ArrayFlat* array = new ( ELeave )
            CDesC16ArrayFlat( KGranularity );
        CleanupStack::PushL( array );

        array->AppendL( path );
    #endif
}

```

```
//This would show the bearer selection:
//iSendUi->CreateAndSendMessagePopupQueryL(
    _L("Send UI"), capabs, NULL, array ),

//This is the actual sending function.
//The selected bearer is given in the aCommand
iSendUi->CreateAndSendMessageL(aCommand, NULL, array);

CleanupStack::PopAndDestroy(); // array
#endif
}
else
{
    //File Selection Canceled by user
    iAppView->LogL(KCanceled);
}
}
```

In the S60 3rd Edition, Feature Pack 1 SDK, there is a messaging example that also demonstrates the usage of the Send UI API in sending SMS, MMS, and e-mail.

8. Security and configurations

This chapter discusses Bluetooth Security Manager and low-level Bluetooth configurations using, for example, the Publish and Subscribe API. Platform security is a separate concept, and it is covered in Section 2.4, "Platform security — effects on Bluetooth application development."

8.1 Bluetooth Security Manager

The Symbian OS API provides a Bluetooth Security Manager. As the name suggests, this component manages the security for Bluetooth connections. It allows a client application to specify a set of security settings. The security settings determine if user authorization, authentication, or encryption is required when establishing a connection. These security settings are applied to a specific server, protocol, and channel. The Bluetooth Security Manager ensures that connections adhere to the client application settings.

Under the original architecture (Bluetooth API v1), security was configured for incoming connections by the method `RBTSecuritySettings::RegisterService`. In Bluetooth API v2, security settings are provided when a listening socket is to be opened using the method `TBTSockAddr::SetSecurity`. This also obviates the need to call `RBTSecuritySettings::UnregisterService` because security settings are automatically removed when the relevant listening socket closes. The same API is also used to set the desired security on outgoing connection sockets. This replaces the `Ioctl`'s `KHCIEncryptIoctl` and `KHCIAuthRequestIoctl`, which in Bluetooth API v2 return `KErrNotSupported`.

The Bluetooth Security Manager is implemented as a server in the Symbian OS API. The API provides the client application with the `TBTServiceSecurity` class to encapsulate security setting information. The `TBTServiceSecurity` class is defined in the `btmanclient.h` header file in Bluetooth API v1 architecture and in `bt_sock.h` in the newer architecture.

The security options are simply set either on or off by passing a Boolean value. The `SetAuthentication`, `SetAuthorisation`, and `SetEncryption` functions all work along these lines. It is also possible to deny all access with the `SetDenied` function. The client application must also specify the service, protocol, and channel to which the security settings will apply.

Each Bluetooth service is identified with a UUID and each protocol by an integer value. The `bt_sock.h` header file contains a short list of available services and protocols. There are changes between versions, so developers should always check these protocols from the SDK they are using; for example, `KAVCTP` has been added in Bluetooth API v2.

8.1.1 Service security in Bluetooth API v2 architecture

In the new Bluetooth API v2 architecture (from Symbian OS v8.0 onwards) the security is configured by the method `TBTSockAddr::SetSecurity` (instead of the previously used `RBTSecuritySettings::RegisterService`). Security settings are now automatically removed when the relevant listening socket closes.

```
User::LeaveIfError(iListener.Open(iSockServ, _L("L2CAP")));
TL2CAPSockAddr addr;
addr.SetPort(KSDPPSM);
TBTServiceSecurity sdpSecurity;
```

```

sdpSecurity.SetUid(KUIdServiceSDP);
sdpSecurity.SetAuthentication(EFalse);
sdpSecurity.SetAuthorisation(EFalse);
sdpSecurity.SetEncryption(EFalse);
sdpSecurity.SetDenied(EFalse);

addr.SetSecurity(sdpSecurity);

User::LeaveIfError(iListener.Bind(addr));
User::LeaveIfError(iListener.Listen(iQueueSize));

```

The same API can be used to set the desired security on outgoing sockets (replacing the `Ioctl`'s `KHCIEncryptIoctl` and `KHCIAuthRequestIoctl`, which now return `KerrNotSupported`):

```

TBTServiceSecurity sec;
sec.SetAuthentication(ETrue);
TBTSockAddr sockaddr;
sockaddr.SetBTAddr(aDevice.BDAddr());
sockaddr.SetPort(aChannel);
sockaddr.SetSecurity(sec);

```

Now an authenticated link can be connected (`sock` is an open Bluetooth `RSocket`):

```

sock.Connect(sockaddr, iStatus);

```

8.1.2 Service security in Bluetooth API v1 architecture

Since the Security Manager is implemented as a Symbian OS server, any client application wishing to use it must first connect to it and establish a session.

The `RBTMan` class provides the connection to the Security Manager server. This class is also defined in the `btmanclient.h` header file.

The `RBTSecuritySettings` class provides the actual security functionality in the Bluetooth API v1 architecture. This class is defined in the `btmanclient.h` header file.

Once the client application has established a connection and session with the Security Manager, the `RegisterService` function allows the application to specify its required security settings. The `UnregisterService` function allows the client application to remove any security requirements. The `RegisterService` and `UnregisterService` functions are asynchronous and require the client application to supply a `TRequestStatus` object. The server uses this object to notify the client application that it has completed its attempt to modify the security requirements. The `TRequestStatus` object contains an integer representing the success or failure of the attempt.

To configure the security settings for an incoming Bluetooth connection:

- Connect to the security server and open a session.
- Set the settings to a `TBTServiceSecurity` object.
- Register the settings.
- Close the session and the connection.

The first task is to establish the connection to the security server and open a session on it.

```

// Connection to the security manager
RBTMan secManager;

```

```
// Security session
RBTSecuritySettings secSettingsSession;

// Define the security of this port
User::LeaveIfError(secManager.Connect());
CleanupClosePushL(secManager);
User::LeaveIfError(secSettingsSession.Open(secManager));
CleanupClosePushL(secSettingsSession);
```

Once the session has been established, the `TBTServiceSecurity` settings object is populated with the required settings. In this example, the security settings apply to an OBEX service using the RFCOMM transmission protocol. The channel, authentication, authorization, and encryption settings have already been passed to the function and are used here.

```
// Security settings
TBTServiceSecurity serviceSecurity(KUIdBTObjectExchangeApp,
    KSolBtRFCOMM, 0);

//Define security requirements
serviceSecurity.SetAuthentication(aAuthentication);
serviceSecurity.SetEncryption(aEncryption);
serviceSecurity.SetAuthorisation(aAuthorisation);

serviceSecurity.SetChannelID(aChannel);
```

The security settings are then registered with the security server.

```
TRequestStatus status;
secSettingsSession.RegisterService(serviceSecurity,
status);
```

The client application now waits for the security server to register the settings. If an error occurs, the application leaves.

```
// wait until the security settings are set
User::WaitForRequest(status);
User::LeaveIfError(status.Int());
```

The client application no longer requires access to the security server, and its session and connection are terminated.

```
//close secManager, secSettingsSession:
CleanupStack::PopAndDestroy(2);
```

8.2 Bluetooth Publish and Subscribe

In the Bluetooth API v1 architecture, there were many different interfaces and methods, such as sending an `ioctl` command with an HCI command, for changing or checking stack and hardware settings. Beginning in Symbian OS v8.0a, dynamic configuration of Bluetooth is now handled using the new Publish and Subscribe API (defined in `bt_subscribe.h`). Note that there are changes in the key names in S60 3rd Edition; see Table 5 for a listing of some of the changes.

8.2.1 Bluetooth categories and property keys

The Bluetooth stack publishes the values of its various settings (properties) in the Bluetooth category. Applications can retrieve the value of a property, set a new value for it (publish), and register (subscribe) for notifications of changes to a property. Applications do not access the property keys in the Bluetooth category directly but they make a request by setting the value of the *equivalent*

property in the Bluetooth *Control* category. The value of the key in the Bluetooth category will be updated when the request is serviced.

Therefore a typical pattern for an application wishing to change a setting is as follows:

1. Define the key in the Bluetooth Control category.
2. Subscribe to the desired key in the Bluetooth category (`KPropertyUIdBluetoothCategory`).
3. Call the `RProperty::Set()` method on the matching key in the Bluetooth Control category (`KPropertyUIdBluetoothControlCategory`).
4. When the stack has serviced the request, it will republish the new value of the key (whether changed or unchanged) on the Bluetooth category, calling the `RunL` of any subscribers.

Retrieving a property value can be done by calling the `RProperty::Get()` method.

The available property keys are listed in Table 5. Note that there is no longer an interface for changing the Page Scan status of the stack. The Bluetooth stack takes care of enabling Page Scan Mode (=scanning for connection requests) automatically when a listening socket is created.

The only general setting not using the Publish and Subscribe mechanism is control of the local Bluetooth device name, which uses the same API as before (`RHostResolver::SetHostName`).

Key	Type	Description
<code>KPropertyKeyBluetoothLocalDeviceAddress</code> In S60 3rd Edition: <code>KPropertyKeyBluetoothGetLocalDeviceAddress</code>	Byte Array	The Local Device Address as a descriptor (BluetoothCategory only).
<code>KPropertyKeyBluetoothPHYCount</code> In S60 3rd Edition: <code>KPropertyKeyBluetoothGetPHYCount</code>	Integer	The number of currently connected Bluetooth Physical Links.
<code>KPropertyKeyBluetoothConnecting</code> Deprecated in S60 3rd Edition, use <code>KPropertyKeyBluetoothGetConnectingStatus</code> instead.	Integer	1 - If actively paging another device (i.e., connecting). 0 - If not paging another device.
<code>KPropertyKeyBluetoothScanning</code> Deprecated in S60 3rd Edition, use <code>KPropertyKeyBluetoothGetScanningStatus</code> instead. There's also a const <code>KPropertyKeyBluetoothSetScanningStatus</code> added in S60 3rd Edition.	Integer	The current scanning mode.
<code>KPropertyKeyBluetoothLimitedDiscoverable</code> Removed in S60 3rd Edition. There are get and set	Integer	Value from the enum <code>THCIScanEnable</code> :

Key	Type	Description
versions for this key.		<code>ENoScansEnabled</code> , <code>EInquiryScanOnly</code> , <code>EPageScanOnly</code> or <code>EInquiryAndPageScan</code> .
<code>KPropertyKeyBluetoothDeviceClass</code> Removed in S60 3rd Edition. There are get and set versions for this key.	Integer	The full integer value of the current Class of Device.
Key <code>KPropertyKeyBluetoothRegistryTableChange</code> Is deprecated in S60 3rd Edition; the get version should be used instead.	Integer	The index of the table that was changed.

Table 5: Bluetooth property keys

8.2.2 Defining keys in the Bluetooth Control category

If the application wants to change any properties it has to define them in the Bluetooth Control category. Defining the keys can be done as follows:

```
TIntr=iProperty.Define(KPropertyUIdBluetoothControlCategory,
    KPropertyKeyBluetoothScanning,
    RProperty::EInt);

if((r!=KErrAlreadyExists))
{
    Use::LeaveIfError(r);
}
```

After the definition is done the value can be set by calling `RProperty::Set()`.

8.2.3 Getting the values

When an application wants to get some values from Bluetooth Publish and Subscribe, `RProperty::Get` is used (or a `RProperty` member variable is used, as in the example above). In the following code the Bluetooth Device Address is read:

```
TPckgBuf<TBTDevAddr> aDevAddrPckg;
//IN V1:
TInt error = RProperty::Get(KPropertyUIdBluetoothCategory,
    KPropertyKeyBluetoothLocalDeviceAddress, aDevAddrPckg);
//IN V2:
TInt error = RProperty::Get(KPropertyUIdBluetoothCategory,
    KPropertyKeyBluetoothGetLocalDeviceAddress, aDevAddrPckg);
```

For more information on Publish and Subscribe, see the S60 SDK help.

8.3 Checking if Bluetooth is supported

The features of the mobile device can be checked using `CFeatureDiscovery` (see `featdiscovery.h` and `featureinfo.h`) from S60 2nd Edition, Feature Pack 3 onwards:

```
TBool isSupported =
CFeatureDiscovery::IsFeatureSupportedL(KFeatureIdBt);
```

8.4 Checking Bluetooth power mode

From S60 3rd Edition (Symbian OS v9.1) onwards, third-party developers can read many common settings, such as the Bluetooth power state, from Central Repository using the Central Repository API:

```
CRepository* crep = CRepository::NewL(KCRUidBluetoothPowerState);
TInt value=0;
User::LeaveIfError( crep->Get(KBTPowerState, value) );
```

The access to a repository is done using the `CRepository` class, which is defined in `centralrepository.h`. The constants `KCRUidBluetoothPowerState` and `KBTPowerState` are in `BTServerSDKCRKeys.h`.

In S60 2nd Edition, `CSettingInfo` (see `settinginfo.h` and `settinginfoids.h`) can be used to check the Bluetooth power mode. If there is a need to turn Bluetooth on, the user can be provided with a dialog (see Section 8.4.1, "Asking the user to turn Bluetooth power on").

```
#include <SettingInfo.h>
#include <aknotewrappers.h>

//NULL because there's no observing functions used
CSettingInfo* settingInfo = CSettingInfo::NewL(NULL);
CleanupStack::PushL(settingInfo);

//getting the current bluetooth power mode

TInt btPowa;
TInt err = settingInfo->Get( SettingInfo::EBluetoothPowerMode, btPowa
);

if (err)
{
    //handle error
}

if (btPowa)
{
    //power is on
}
else
{
    //power off
    // Do something to switch BT on,
    // e.g. show notifier, see next chapter
}
CleanupStack::PopAndDestroy(settingInfo);
```

8.4.1 Asking the user to turn Bluetooth power on

When device discovery is initiated using the Bluetooth UI (discussed in Section 3.1, "Using the Bluetooth UI"), the power status is checked automatically, and if Bluetooth is off, the user is provided with a dialog to switch it on. This is not automated in all cases: the server-side application always has to take care of the Bluetooth power check; it is also the case for the client side if `RHostResolver` is used for device discovery (see Section 3.2, "Running a device discovery with `RHostResolver`").

A Bluetooth power check can be done using the Bluetooth Notifier API. This API was introduced in S60 2nd Edition, Feature Pack 1 (`btnotifierapi.h` in the S60 3rd Edition SDK and `btnotif.h` in the S60 2nd Edition SDKs supporting

FP1, FP2, and FP3). Prior to S60 2nd Edition, Feature Pack 1, the constant value has to be used directly:

```
const TUid KPowerModeSettingNotifierUid = {0x100059E2}
```

For showing the dialog, use the following function from the `RNotifier` class:

```
StartNotifierAndGetResponse(TRequestStatus &aRs, TUid  
aNotifierUid, const TDesC8 &aBuffer, TDes8 &aResponse);
```

Bluetooth power-check functionality has been implemented in the Bluetooth PMP Example [5].

9. Terms and abbreviations

Term or abbreviation	Meaning
API	Application programming interface. A definition (header file) of a class and related functions and constants to be used by applications.
Authentication	Method of checking the identity of the remote device by comparing the stored link keys.
Authorization	User-level confirmation to the incoming connection request.
Bluetooth	A wireless technology used for sending data and audio.
Bluetooth API v2	Term used when speaking about Bluetooth APIs since Symbian OS v8.0a.
DLL	Dynamic-linked library. An application (or part of it) that does not have a user interface.
Encryption	Protecting the data from eavesdropping.
GIAC	Generic Inquiry Access Code.
GUI	Graphical user interface (display part).
HCI	Host Controller Interface.
LIAC	Limited Inquiry Access Code.
L2CAP	Logical Link Control and Adaptation Protocol.
LMP	Link Manager Protocol.
Master	Bluetooth role; the device that initiates the connection.
OBEX	Object Exchange (protocol).
OS	Operating system.
Piconet	Bluetooth ad-hoc network; one master and up to seven active slaves.
Point-to-point	A connection between a master and a slave.
PMP	Point-to-multipoint. Simultaneous connections from one master to multiple (up to seven) slaves.
RFCOMM	Serial port emulation protocol.
Scatternet	Scenario where one device participates in two piconets.
Slave	Bluetooth role; the device that accepts the connection.
SDP	Service Discovery Protocol.
UI	User interface.
UUID	Universal Unique Identifier.

10. References

- [1] *Bluetooth Core Specification*,
https://www.bluetooth.org/apps/content/?doc_id=44515 (requires registration).
- [2] *Bluetooth Technology Overview*, available at www.forum.nokia.com.
- [3] *Chat Example*, available in the S60 3rd Edition SDKs.
- [4] *S60 Platform: Bluetooth OBEX Example*, available at www.forum.nokia.com
<http://www.forum.nokia.com/>.
- [5] *S60 Platform: Bluetooth Point-to-Multipoint Example*, available at
www.forum.nokia.com.
- [6] *S60 Platform: Identification Codes*, available at www.forum.nokia.com.

Appendix A. Bluetooth API v2 changes

Although not every single change is listed, many of the important Bluetooth API changes discussed throughout the document are summarized in this appendix. These listings are especially useful for developers creating applications for multiple S60 releases. Table 6 presents the major changes in Bluetooth API v2 architecture (since Symbian OS v8.0a); Table 7, in Appendix B, lists the (mostly minor) changes in S60 3rd Edition (Symbian OS v9.1).

Table 6: Changes in Bluetooth API v2 architecture

Module	Change in Bluetooth API v2
Bluetooth Registry API	<p><code>btregistry.h</code> was removed and registry operations are handled with new classes defined in <code>btmanclient.h</code>:</p> <pre> RBTRegServ RBTRegistry TBTRegistrySearch CBTRegistryResponse </pre> <p><code>btdefcommport.h</code>, which manages port and service settings of the legacy serial port applications, was removed.</p>
Low-level Bluetooth configurations	<p>Earlier updating of general Bluetooth device settings (e.g., checking the device address or changing the scanning mode) was handled in an ad-hoc manner using various methods and <code>Ioctl</code>s in different APIs. The dynamic configuration of Bluetooth is now more unified and handled using the new Publish and Subscribe API (defined in <code>bt_subscribe.h</code>).</p> <p>To set the inquiry scanning mode, the method <code>RProperty::Set</code> should be called instead of sending <code>KHCIWriteScanEnableIoctl</code> down the socket.</p> <p>There is no longer any need for an interface to change the Page Scan status of the stack, as this is now solely managed by the stack (i.e., when a listening socket is created the stack will enable Page Scan Mode).</p>
Bluetooth sockets	<p>The method for the listening device to specify the listening channel, <code>RSocket::SetLocalPort(aPortNumber)</code>, defined in <code>es_sock.h</code> (provided by Symbian OS), is deprecated. Applications should instead call the <code>RSocket::Bind(sockAddr)</code> method (defined in <code>es_sock.h</code>). The constants for the RFCOMM or L2CAP ports are defined in <code>bt_sock.h</code>. <code>RSocket::LocalPort</code> should be used to get the port.</p> <p>The methods of creating and manipulating SCO connections via <code>Ioctl</code>s on connected RFCOMM sockets have been deprecated and replaced with <code>CBluetoothSynchronousLink</code> (defined in <code>bt_sock.h</code>).</p>

Module	Change in Bluetooth API v2
Bluetooth device	Previously, the Bluetooth device name was stored in the <code>CBTDevice</code> class as a Unicode descriptor. The Bluetooth device name length has changed: it is now stored as an 8-bit (UTF-8 encoded) descriptor. See Section 6.2, "Device name length."

Appendix B. Bluetooth API changes in S60 3rd Edition

Table 7 shows minor changes in Symbian and S60 Bluetooth APIs, introduced in S60 3rd Edition.

Table 7: Changes in S60 3rd Edition

File or class	Change in S60 3rd Edition
bt_header.h	Removed.
bt_sock.h	<p>A number of changes in Bluetooth sockets:</p> <p>Many constants have been added, such as <code>KSolBtAVDTPSignalling</code>, <code>KSolBtAVDTPMedia</code>, <code>KSolBtAVDTPReporting</code>, <code>KSolBtAVDTPRecovery</code>, and <code>KSolBtAVDTPInternal</code>. Some constants have also been removed.</p> <p>Classes <code>TBTSyncBandwidth</code> and <code>TBTteSCOLinkParams</code> added.</p> <p>Class <code>CBluetoothSynchronousLink</code> has new functions:</p> <pre>TInt SetupConnection(const TBTDevAddr& aDevAddr, const TBTSyncPackets& aPacketTypes); TInt AcceptConnection(const TBTSyncPackets& aPacketTypes); void SetCoding(TUint16 aVoiceSetting); void SetMaxBandwidth(TBTSyncBandwidth aMaximum); void SetMaxLatency(TUint16 aLatency); void SetRetransmissionEffort(TBTteSCORetransmissionTypes aRetransmissionEffort); TUint16 Coding(); TBTSyncBandwidth Bandwidth(); TUint16 Latency(); TUint8 RetransmissionEffort();</pre> <p>There are also source code brakes. Functions <code>Socket()</code> and <code>ListeningSocket()</code> have been removed and replaced with:</p> <pre>RSocket& SCOSocket(); RSocket& ESCOSocket(); RSocket& ListeningSCOSocket(); RSocket& ListeningESCOSocket();</pre> <p>Functions <code>HandleSetupConnectionCompleteL</code> and <code>HandleAcceptConnectionCompleteL</code> have now a second parameter, <code>TSCOType aSCOType</code>.</p> <p>There are new functions in classes <code>TBTAccessRequirements</code> and <code>TBTServiceSecurity</code>:</p> <pre>TUint PasskeyMinLength() const; TInt SetPasskeyMinLength(TUint aPasskeyMinLength);</pre> <p>In <code>TBTOptions</code> enumeration, <code>KBTRegisterCodService</code> has been added.</p>

File or class	Change in S60 3rd Edition
<code>bt_subscribe.h</code>	<p><code>const TUint KPropertyKeyBluetoothLocalDeviceAddress</code> has been deprecated; use <code>KPropertyKeyBluetoothGetLocalDeviceAddress</code> instead. The same is true for <code>KPropertyKeyBluetoothPHYCount</code> and <code>KPropertyKeyBluetoothSetScanningStatus</code>, among many other constants.</p>
<code>btcmtdm.h</code>	Not present in S60 3rd Edition but it is in S60 3rd Edition FP1.
<code>btdevice.h</code>	<p>Type definition <code>CBtSecurityArray</code> has been removed.</p> <p>The <code>TBTDeviceSecurity</code> class contains a new constructor and functions for security key length handling.</p> <p><code>TBTNamelessDevice</code> contains new functions:</p> <pre>TBool IsPaired() const; void SetPaired(TBool aPaired); TBool IsValidPaired() const; const TBTPinCode& PassKey() const; void SetPassKey(const TBTPinCode& aPassKey); TBool IsValidPassKey() const; TUint PassKeyLength() const;</pre> <p>Also, many earlier deprecated functions have been removed.</p>
<code>bttextnotifiers.h</code>	<p><code>TBTBasebandNotificationParamsRemoved</code> has been removed.</p> <p>The new class <code>TBTDeviceList</code> has been introduced.</p>
<code>btmanclient.h</code>	<p>Class <code>TBTAuthorisationParams</code> has been removed and replaced with new classes <code>TBTAuthorisationParams</code> and <code>TBTNotifierParams</code>.</p> <p>Class <code>RBTSecuritySettingsB</code> has been removed along with other B-ending classes, which were only kept for BC reasons.</p>
<code>btnotifierAPI.h</code>	<code>btnotifierAPI.h</code> was introduced in S60 3rd Edition and replaced <code>btnotif.h</code> .
<code>btsgdp.h</code>	<p>New constants have been added concerning capabilities, features, and audio.</p> <p>Class <code>CSdpAttrValueUint</code> has new functions:</p> <pre>void Uint64(TUint64& aValue); void Uint128(TUint64& aLo, TUint64& aHi);</pre> <p>Class <code>SdpUtil</code> has been introduced.</p>
<code>btserverSDKcrkeys.h</code>	Added (Central Repository keys).

File or class	Change in S60 3rd Edition
<code>bttypes.h</code>	<p>Constant <code>KMinESCOLatency</code> has been added among others. There is a new type definition <code>TBTLinkModeSet</code>. Class <code>TBTSyncPackets</code> has been introduced.</p> <p><code>TUUID</code> now has <code>FixedLength()</code> and <code>MinimumSize()</code> functions.</p> <p><code>TL2CapConfig</code> has new functions such as <code>TInt ConfigureReliableChannel(TUint16 aRetransmissionTimer)</code> and <code>TInt ConfigureChannelPriority(TChannelPriority aPriority)</code> to be used instead of <code>SetupXXX()</code> functions, which have been deprecated.</p>
<code>hcibase.h</code>	<p>Many new constants have been added. The <code>CHCIBase</code> class has some new functions.</p>
<code>hcitypes.h</code>	<p><code>TLinkType</code> enumeration has new value <code>EeSCOLink</code>. New enumeration <code>TAirMode</code> has been added along with class <code>TBTSyncConnectOpts</code>.</p>
<code>sendui.h</code>	<p>The SendUI API has been redesigned: <code>CSendUi</code> has replaced <code>CSendAppUi</code> (see Section 7.6, "Easy way of using OBEX with Send UI").</p>

Evaluate this resource

Please spare a moment to help us improve documentation quality and recognize the resources you find most valuable, by [rating this resource](#).