

Java Bluetooth Object Exchange



Alan Newman

Sensible Development Ltd

Version: 1.7

1	INTRODUCTION	2
	1.1 WHY IS THIS PAPER IMPORTANT TO SYMBIAN OS DEVELOPERS?	2
	1.2 PAPER CONTENT OVERVIEW	2
	1.3 RESOURCES AND REFERENCES THIS PAPER USES	2
2	OBJECT EXCHANGE (OBEX) BACKGROUND.....	3
	2.1 INTRODUCTION TO OBEX	3
	2.2 EXCHANGE OF OBJECTS SUCH AS vCARDS, vNOTE AND vCAL OBJECTS.....	3
3	OBJECT EXCHANGE USING JSR 82	3
	3.1 BRIEF EXPLANATION OF THE JSR 82	3
	3.2 GENERAL STRUCTURE OF THE BLUETOOTH API.....	4
	3.3 EXPLANATION OF OBEX API WITHIN JSR 82	5
	3.4 SAMPLE CODE TO DEMONSTRATE HOW THE OBEX WORKS USING JSR 82.....	6
4	SYMBIAN OS v7.0S PHONES	7
	4.1 USE RFCOMM	8
	4.2 OPEN SOURCE	8
5	SAMPLE PROGRAM.....	9
	5.1 THE OBJECT EXCHANGE PROCESS	9
6	CONCLUSION	15
7	TABLES	17
	7.1 OBEX OPERATION CODES	17
	7.2 OBEX RESPONSE CODES	17
	7.3 CONNECT REQUEST FORMAT	18
	7.4 OBEX REQUEST FORMAT	18
	7.5 SERVER RESPONSE FORMAT.....	19
	7.6 A PUT OBEX EXAMPLE	19

1 Introduction

1.1 Why is this paper important to Symbian OS developers?

This paper addresses a limitation within Symbian's JSR 82 implementation. JSR 82 is a set of Java APIs to allow Java enabled devices to integrate into a Bluetooth environment. The APIs contains two packages: `javax.bluetooth` and `javax.obex`. However, Symbian OS v7.0s introduced support for `javax.bluetooth`, but not the `javax.obex` package.

Documentation for developers wishing to find alternatives to this absence is patchy and this is due in part to the relatively low number of Bluetooth-enabled Java Symbian OS phones out in the market. By providing a paper directed specifically at Symbian developers covering both the issue of sourcing references and also providing a solution to the limitations within the Symbian OS 7.0s implementation, it will become a valuable source of information.

1.2 Paper content overview

This paper will address how Symbian OS Java developers can extend the functionality of Java MIDP applications developed for Symbian v7.0s phones with the use of low level protocols to carry out object exchange (OBEX) between a Java MIDP application and another phone or indeed a PC. This exchange can be executed without the need for a MIDP 2.0 server application on the recipient phone or machine. This will further extend the uses of MIDP 2.0 applications and may provide an extra layer of functionality such as the propagation of a viral marketing campaign.

Without MIDP applications being able to use OBEX, they are restricted to only communicating with other devices, paired by a common application. Currently this would be best achieved with both phones executing the application in a client/server relationship. Whilst this is useful, especially in mobile gaming, it does not help the promotion of the application and in the long term reduces marketing capabilities. This leaves Symbian MIDP applications only be able communicate with other MIDP applications on other devices via Bluetooth.

This paper therefore will focus on explaining how this functionality might be created and provides a sample MIDlet to demonstrate how a developer might provide a solution. It also pulls together a number of resources, providing a coherent approach for developers wishing to develop MIDP 2.0 OBEX applications. Details of these will be provided in the next section.

1.3 Resources and references this paper uses

This paper pulls together a number of resources concerning OBEX communication with a MIDP 2.0 application. They are as follows:

- IrDA Object Exchange Protocol documentation (Chapter 12) (<http://www.irda.org>)
- AvetanaBluetooth JSR-82 implementation - (<http://sourceforge.net/projects/avetanabt/>)
- Benhui.net (<http://www.benhui.net>) – application developer resources website.
- [“Programming Java 2 Micro Edition on Symbian OS” by Martin de Jode](#) – background to Bluetooth MIDP 2.0 application architecture
- Java Community Process (JSR 82) - <http://www.jcp.org/en/jsr/detail?id=82>

2 Object Exchange (OBEX) Background

2.1 Introduction to OBEX

Before we move on, it would be good at this point to understand a little more how OBEX came to be and how it works. This will help us understand just how we have managed to get OBEX working with the need for the OBEX API.

The object exchange protocol (OBEX) enables computers to interact with each other over personal area networks (PAN). Originally specified for Infra Red communication (www.irda.org), it has also been adopted as a means to exchange data objects over Bluetooth.

It is similar in characteristic to the HTTP protocol. Methods are provided where additional information can be exchanged between client and server. Unlike HTTP, where these headers are strings, the data is passed as byte arrays or byte sequences. These headers include length, name and description type. They are not mandatory in use, but will help the client and server to know more about the data that is being exchanged. This helps particularly during the detection of errors or interruptions in transmission.

An OBEX session is established by portable devices, such as PDAs, Symbian phones and cameras, with a CONNECT request and end with a DISCONNECT. During the session the client can GET objects from the server or PUT objects to the server. These objects can be sent in one go or can be broken into a series of packets and sent over the course of the session. As each packet is sent, it waits for an acknowledgement of successful transmission before sending the proceeding packet. This cycle continues until either an error occurs, the object is sent in entirety in the case of a PUT, or is received intact with a GET request.

2.2 Exchange of objects such as vCards, vNote and vCal objects

So, how is this useful to the Symbian mobile developer? Well, it is interesting because it gives Java developers the ability to create mobile applications that can send objects to other mobile phones. This is useful in a number of ways. It allows them to utilise MIDI etc for viral campaigns, or perhaps to send multimedia objects from one phone to another. If OBEX is used along with other messaging protocols such as SMS, it provides for a very effective marketing tool.

So, what can we send?

vCards, vNotes and multi-media objects can be sent. The objects are constructed within the MIDI etc and sent to the remote phones. It also means we can send objects to other Bluetooth enabled computers as well, such as PCs and servers. All they need to be able to do is to have an OBEX server running to receive the inbound requests.

3 Object exchange using JSR 82

3.1 Brief explanation of the JSR 82

JSR 82 is an optional package for MIDP implementation. This means that it may not have been included in the Java MIDP 2.0 implementation. JSR 82 comprises two parts, the Bluetooth API, which provides an interface to the device's Bluetooth transport layer and the OBEX API, which provides for object exchange between devices.

Java MIDP for Symbian OS v7.0 phones has been implemented without the OBEX API, whereas the Bluetooth API is fully functional.

Bluetooth wireless technology operates over its own specific protocol stack. This stack is composed of four layers all providing different functionality. OBEX is an adopted protocol that forms a part of this stack, and was specified, originally for use over IrDA. The four layers are as follows:

a) Bluetooth Core Protocols.

This layer is made up from the following:

- Baseband layer which provides the physical connection between the two devices
- Link Manager Protocol (LMP), which manages security and the link set up between the Bluetooth devices
- L2CAP – this multiplexes the logical connections made by the upper layer of the protocol
- SDP, which queries for remote device information such as available services and device location

b) Cable Replacement Protocol.

- RFCOMM, which provides cable replacement by the emulation of the RS 232 control and data signals over the Bluetooth baseband

c) Telephony Control Protocol.

- TCS binary, which defines call control signalling for voice and data calls over Bluetooth

d) Adopted Protocols.

- These protocols include PPP, UDP/TCP/IP, OBEX and WAP. OBEX is the more notable protocol in the context of this document

The Bluetooth Special Interest Group has defined a group of profiles, which define the standard ways in which the above protocols and their features should be used. The four “generic” profiles are:

- Generic Access Profile (GAP).
- Serial Port Profile (SPP).
- Service Discovery Application Profile (SDAP).
- Generic Object Exchange Profile (GOEP).

The use of these protocols and profiles is essentially why OBEX works over Bluetooth. It is an adoption of the cable serial port protocol RFCOMM.

3.2 General structure of the Bluetooth API

The API created under the guidance of the Java Community Process, JSR 82, includes two APIs, one for Bluetooth and the other for OBEX. Overall it is split into three functional categories.

- Discovery – concerned with browsing for devices within range, discovering their services and then registering with those services
- Communication – establishing connections with those discovered devices, for communication between Bluetooth enabled applications
- Device management – the management and control of these established connections

As mentioned above, the functionality has been spread over two optional MIDP APIs, which have been broken down into Bluetooth and OBEX as follows:

- i. javax.bluetooth
- ii. javax.obex

In our example toward the end of this paper, we shall be making use of the first package to manage the device discovery, service registration and communication. However, we are unable to make use of the second package, as the Symbian OS 7.0s does not implement this optional package.

The API, as a whole, interfaces with the BCC (Bluetooth Control Center) to manage the security aspects of the device connections with other computers or mobile phones. It also manages any potential conflict between applications as they try to claim use of the Bluetooth resources on the device. The JSR 82 specification provides further details on how this works.

Bluetooth programming is in essence about the client/server model. Applications, once executed, register themselves with their own device Service Discovery Database (SDDDB). Client applications will then browse devices within range and look to make a connection to them and use the resource made available to them.

In terms of MIDP development this might mean a MIDP application has to be present on both client and server mobile phones. For business and gaming applications, this means communication can be established, assuming both phones are equipped with the MIDP application, before they can exchange data. In most cases this data would be only useful and understood by those applications. Data such as high scores or perhaps business contact information could be swapped between applications.

Usually the purchase of the application itself forms the revenue stream for the developer or publisher. If that application can provide some degree of viral marketing, then its popularity is likely to grow. If the client does not possess the MIDP application then they cannot exchange data. However, if the client was to send generic data that the client could understand, then there is the possible means with which super distribution can be achieved.

Therefore, if we can emulate OBEX from within a MIDP application, we can at the very least begin to pass on vCards and vNotes. From there a user could make an SMS request for a link to be sent, from which the application could be downloaded. The user would receive the download location in a premium rate SMS, thus generating revenue, and the application's distribution will have been proliferated.

Of course we do not only have to be concerned with distribution, but also backing up of contact data. We might, for example, want to send contact information to a PC within the mobile phone's PAN. Other uses might also include the back up of contact information to a PC.

3.3 Explanation of OBEX API within JSR 82

As we saw in section 2.1, OBEX is carried out over the course of a session created when two Bluetooth devices, whether they are phones or PCs or any other Bluetooth enabled device, exchange data.

The object of the OBEX API is to wrap the functionality of the native Bluetooth functionality into an interface easily accessible by the developer.

The API therefore provides a number of packages that make this possible. These are used in tandem with the Bluetooth API, which performs the service discovery and registration between the two connected devices.

In section 2.1 we examined the generic process when transferring data from one device to another. So how do we do this using the OBEX API? The following provides a brief overview of the main classes that can be used if the OBEX API was present on the Bluetooth device.

3.3.1 Establishing a connection

Using the Generic Connection Framework, `Connector.open()` is used to initiate the Bluetooth connection. This returns a `javax.obex.ClientSession`, which is used to establish the connection itself.

A `javax.obex.HeaderSet` is created in one of two ways. The `createHeaderSet()` method of `ClientSession` can be invoked or a `null HeaderSet` can be passed to the `ClientSession's connect()` method. In either case a `HeaderSet` is returned.

3.3.2 Header Manipulation

Once the connection has been established, the `HeaderSet` object should be examined to make sure it was correctly set up. The `getResponseCode()` method will return its state and, if `OBEX_HTTP_OK` is returned, then the process should continue. Otherwise proceed to disconnection.

Assuming we are making a PUT, the `HeaderSet` object allows us to set the headers we wish to pass to the server object. It means we can tell the server what we are sending, what it is called and how large it is. This leads a more graceful exchange of objects and allows us to check that the operation was successful.

Typically we might want to tell the server the `NAME` of the object we are sending, its `TYPE` and its `LENGTH`. This is achieved by using the `setHeader()` method of `HeaderSet`.

3.3.3 Exchanging Objects

Now that we have prepared the server for the receipt of our object, we can proceed with its transmission. We will be using a PUT to send the object.

In this case we use the `ClientSession` method `put()` to send the headers to the server. The method is called with the `HeaderSet` as the parameter and it returns a `javax.obex.Operation` object. The same object is also returned when a `get()` is made.

The `Operation` provides us with an `OutputStream` that is used to write and flush the data to the server. We pass our payload as a byte array to the `write()` method before issuing a `flush()`.

3.3.4 Disconnection

If at any point there is need to terminate the connection, either because of an error or upon completion of the operation, then a `disconnect()` and `close()` should be issued to free up the `ClientSession` resource, along with a `close()` on each of the `Operation` and `OutputStream` objects.

3.4 Sample code to demonstrate how the OBEX works using JSR 82

After our brief explanation in the previous subsection, it would be a good idea to flesh out the theoretical steps needed to achieve a successful PUT operation.

In this example we will make the assumption that we can discover the device we wish to push our vCard to. We do cover this in greater detail in our non-OBEX API version in section 5. So we will assume at this point we have managed to discover the remote server to which we wish to send our data.

```
public void doPut(String url, byte[] message)
{
    HeaderSet headerSet = null;
    Operation operation = null;
    OutputStream output = null;
    ClientSession connection = null;

    try
    {
        // open a client connection according
        // to the url provided
        connection = (ClientSession) Connector.open(url);

        // create a set of headers
        headerSet = connection.connect(null);

        // let's tell the server what we are sending to it
        // a vCard, its name & length.
    }
}
```

```

headerSet.setHeader(HeaderSet.LENGTH,
new Long(message.length));
headerSet.setHeader(HeaderSet.NAME, "vcard.vcf");
headerSet.setHeader(HeaderSet.TYPE, "text/vCard");

// check that everything is OK
int responseCode = head.getResponseCode();

if (responseCode != ResponseCodes.OBEX_HTTP_OK)
{
    // deal with a bad response
    if (connection != null)
    {
        try
        {
            connection.disconnect();
            connection.close();
        }
        catch (IOException ioe) {}
        return;
    }
}

// the PUT request
operation = connection.put(headerSet);

// open the output stream
output = operation.openOutputStream();

// send the message
output.write(message);
output.flush();

// all complete!
}
catch(IOException e) {} // deal with this failure

finally
{
    // release the resources here.
    if (out != null)
    {
        output.close();
    }
    if (operation != null)
    {
        operation.close();
    }
}
}

```

4 Symbian OS v7.0s phones

The `javax.obex` package was not introduced with Symbian OS v7.0s. This means object exchange cannot be conducted over the OBEX protocol `btgoep`. Therefore an alternative solution has to be sought over a lower level protocol. The following suggests two possible alternatives:

4.1 Use RFCOMM

One approach is to make use of RFCOMM, which is the Bluetooth serial port protocol (BTSP), rather than Bluetooth over OBEX (BGOEP).

By using the device discovery to determine the address of the remote phone, we could then convert the URL of that device from a BTGOEP to a BTSP device address. We can then connect to the phone over the BTSP protocol to send the OBEX object via Bluetooth.

As we explained in section 2.1, the object exchange is achieved by a series of requests and responses. At the lowest level the communication is carried out by a series of operation codes - see table 7.1 - that are squirted down a stream set up between the client and server. Each request is responded to by one of the response codes in table 7.2. The format in which these operational commands are sent is specified in table 7.4, whilst their responses are outlined on table 7.5.

So how might we actually approach this? The following steps provide a high level view of the actions required:

- a) Browse for Bluetooth devices using the `javax.bluetooth.DiscoveryListener` interface
- b) Create a list of Bluetooth phones that are available to connect to.
- c) Obtain the URL to the desired device.
- d) Take the URL, which will be formatted as BTGOEP protocol and convert it to the BTSP protocol.
- e) Connect to the device by creating a `StreamConnection`. A typical `Connection` request should take the form of that specified in table 7.3.
- f) Create a byte array for the headers and `write()` them to the stream.
- g) Read the response from the server application.
- h) Create a byte array for the object payload and `write()` that to the stream.
- i) Read the response from the stream to check for successful transmission.
- j) Either continue transmission or disconnect the session.

This approach will work quite nicely, but the main drawback is that it means initially the management of the connection and header interaction with the server will have to be 'hand-rolled'.

At the lowest level, the interaction between client and server to send a `vCard` to a Bluetooth server would, according to the IrDA specification, take the form of the information in table 7.6. The example in that table steps through the client/server interaction required to send the bytes required to the server application. These bytes would be sent down the `StreamConnection` as a byte array. Upon each transmission, the server will respond with a success or failure code.

4.2 Open Source

Of course, creating our own package to deal with OBEX is more than possible. There are developers who have done the hard work for us and it will speed up development time to allow us to concentrate on other tasks.

So perhaps it is worth looking at one particular open source package that does the job for us. The `AvetanaBluetooth` package (<http://sourceforge.net/projects/avetanabt/>) provides us with a number of classes that mirror the JSR 82 OBEX API by using the RFCOMM protocols to do the transfer of the data across between the two Bluetooth devices.

Our sample code, in section 5 and also provided in full with this paper, provides a fuller explanation of how the package should be used. Before we examine the detailed solution, first we should briefly consider the features offered by the package as it emulates the functionality of the OBEX API.

- a) Provision of mirror objects such:
 - i. ClientSession
 - ii. HeaderSet
 - iii. Operation
 - iv. Authenticator
 - v. PasswordAuthentication
 - vi. ResponseCodes
 - vii. ServerRequestHandler
 - viii. SessionNotifier
- b) Provision of a connection object in the form of OBEXConnection
- c) MD5 provision

The Aventana package mimics the OBEX API and provides us with an interface to carry out the tasks we have described in section 4.1. The API basically gives us a wrapper with which to communicate over the Bluetooth connection to any Bluetooth server within range.

In the following section we will examine more closely how to put these classes to good use, by providing a fuller discovery, server registration and connection between a MIDP application and any Bluetooth enabled server device.

5 Sample Program

The best way of demonstrating how this all fits together, is of course to provide an example application. We have put together a small functional MIDlet that simply allows us to transfer a vCard from one phone to another using Bluetooth as the transport mechanism.

The object could of course also be a vCalendar or vNote object. Unfortunately it is outside the functionality of Bluetooth to be able to send SMS, which does of course prevent us from being able to send parseable URL from one phone to another. Otherwise this would be a useful viral function.

5.1 The object exchange process

The architecture of our sample application reflects the areas of functionality we will describe below.

We have used an open source package to do part of the work for us, whereas the rest is based upon code previously outlined in "[Programming Java 2 Micro Edition on Symbian OS](#)" by Martin de Jode (Symbian Press).

A listener, `DiscoveryListener`, which is found in the `javax.bluetooth` package of the JSR 82, manages the discovery of Bluetooth devices and their services. This listener provides interface methods that are called upon the completion of each step in the discovery of devices and services.

5.1.1 Device discovery

a) Location of the remote phones in range

In order to find which services we are able to connect to, we first need to see which phones are within range.

This process is managed by the `DiscoveryListener`, which provides an interface and a number of methods to handle the various stages of this process. We have contained the phone location part of this of this process in `BluetoothDeviceDiscovery.java`.

Initially the `MI D I e t` initialises the `B I u e t o o t h D e v i c e D i s c o v e r y` object, passing itself as a parameter. This helps us pass information back to the user via the interface. It also helps provide access to the `MI D I e t`'s call-back methods once the device search has been finalised.

The device discovery object is initialised as follows:

```
public BluetoothDeviceDiscovery(SymbianBluetoothMIDlet midlet)
{
    this.midlet = midlet;

    try
    {
        LocalDevice = LocalDevice.getLocalDevice();
        LocalDevice.setDiscoverable(DiscoveryAgent.GIAC);
        agent = LocalDevice.getDiscoveryAgent();

        midlet.setAlertFormGauge(2);
        devicesDiscovered = 0;
    }
    catch(Exception e)
    {
        midlet.setAlertMessage("Bluetooth communication error. Please try again.");
    }
}
```

The initialisation provides us with a local device object and sets the type of discovery agent for when the search is made.

We can search for devices in one of two modes, GIAC and LIAC. The GIAC mode allows us to browse for all devices in the vicinity, where as LIAC discovers only remote devices.

Once we have initialised the `B I u e t o o t h D e v i c e D i s c o v e r y` object we need to commence the search. This method, `startDeviceSearch()`, is called from the `MI D I e t`.

```
public void startDeviceSearch()
{
    try
    {
        midlet.setAlertFormGauge(4);
        agent.startInquiry(DiscoveryAgent.GIAC, this);
    }
    catch(Exception e)
    {
        midlet.setAlertMessage("Bluetooth communication error. Please try again.");
    }
}
```

This searches for all the devices in the vicinity and returns a user friendly name which can be used for display to the user via the user interface. We also update the progress bar to manage the users expectation of how far through the process we are.

As the phones are found they are added to a vector that is used by the user interface class. This method is automatically called by the interface `DiscoveryListener` as phones are found.

```
public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod)
{
    //add the remote device to the collection
    remoteDevices.addElement(btDevice);

    // inform the user of progress
    devicesDiscovered++;
}
```

```

        miDlet.setAlertMessage("Devices found: "+devicesDiscovered);
    }

```

As the phones are added to the `remoteDevices` vector, we also increment the `devicesDiscovered` value so that we can update the user interface. This shows the user that something is happening and that progress is being made.

Another method also called by the listener interface is the `inquiryCompleted()` method, which is called once the search has been exhausted. The user interface is updated and the `Vector` is converted to an array of devices and then passed back to the `MI Dlet`'s call back method.

```

public void inquiryCompleted(int discType)
{
    miDlet.setAlertFormGauge(7);
    RemoteDevice[] devices = null;

    if(remoteDevices.size()!=0)
    {
        devices = new
            RemoteDevice[remoteDevices.size()];

        for(int i = 0; i < remoteDevices.size(); i++)
        {
            devices[i] = (RemoteDevice)
                remoteDevices.elementAt(i);
        }
    }

    miDlet.setAlertFormGauge(8);
    miDlet.inquiryCompleted(devices);
}

```

The `MI Dlet` then updates the Bluetooth UI Form and shows the user which phones have been found.

b) Display the phones to the user

These phones are now available to be chosen as a recipient for our vCard. This is carried out by the `inquiryCompleted()` method.

```

public void inquiryCompleted(RemoteDevice[] devices)
{
    setAlertFormGauge(9);
    setAlertMessage("Search completed");

    try
    {
        if (devices!=null)
        {
            this.remoteDevices = devices;
            String[] names = new String[devices.length];
            address = new String[devices.length];

            for(int i = 0; i < devices.length; i++)
            {
                try
                {
                    String name = devices[i].getFriendlyName(false);
                    address[i] = devices[i].getBluetoothAddress();
                    names[i] = name;
                }
                catch(Exception ioe)

```

```

        {
            showAlert(i oe. toString());
        }
    }

    // create & display the Bluetooth device List
    bluetoothUI = new BluetoothUI ();
    backmain = new Command("Back", Command.SCREEN, 2);
    sendBluetooth = new Command("Send", Command.SCREEN, 2);
    bluetoothUI . populateList(names);
    bluetoothUI . addCommand(backmain);
    bluetoothUI . addCommand(sendBluetooth);
    bluetoothUI . setCommandListener(this);
    display.setCurrent(bluetoothUI);
}
else
{
    showAlertMessage("No bluetooth devices were found");
}

}
catch(Exception e) {
    showAlertMessage(e.toString());
}
}

```

The MI Dlet call-back takes the array of devices and uses it to populate the BluetoothUI object. Stored within each device object is some useful information, namely a URL to the remote phones that we can use later to connect other phone's services.

It is now down to the user to select the phone they wish to communicate with.

c) Capture user selection, then proceed to service discovery

The user navigates the list on screen and from here they can select which phone to send a vCard or other type of object to.

When they select the phone, we use this to locate the Bluetooth address of the phone, and convert this to a URL.

This process is instigated by a command on the forms. The pertinent part of the code is as follows:

```

// retrieve the selected device from the device list in the UI/form
int index = bluetoothUI.getSelectedDevice();
remoteDevice=remoteDevices[index];

// commence the service search on the desired device.
btServices.startServiceSearch(remoteDevices[index]);

```

We locate the selected phone from the device array and then pass its value as a RemoteDevice object to the BluetoothServiceDiscovery object, which is also implemented as a DiscoveryListener object interface.

5.1.2 Service discovery

The DiscoveryListener interface object, BluetoothServiceDiscovery has been previously created with some initialisation objects.

Again we maximise the search to encompass all devices in the vicinity by using the GIAC option for searching.

We get the device itself, and then create a discovery agent. This agent is later used by the `startServiceSearch()` method to look for available services on the chosen remote phone.

```
public BluetoothServiceDiscovery(SymbianBluetoothMIDlet midlet)
{
    this.midlet = midlet;
    try
    {
        LocalDevice localDevice = LocalDevice.getLocalDevice();
        localDevice.setDiscoverable(DiscoveryAgent.GIAC);
        agent = localDevice.getDiscoveryAgent();
    }
    catch(Exception e)
    {
        midlet.setAlert("Bluetooth communication error. Please try again.");
    }
}
```

a) Obtaining remote phone service record

Once we have located the available service on the remote phone we need to establish what the service record is for that phone.

```
public void startServiceSearch(RemoteDevice remoteDevice)
{
    try
    {
        String device = remoteDevice.getFriendlyName(true);
    }
    catch(IOException ioe)
    {
        midlet.setAlert(ioe.toString());
    }

    try
    {
        agent.searchServices(attrSet, uuidSet, remoteDevice, this);
    }
    catch(Exception e)
    {
        midlet.setAlert("Bluetooth communication error. Please try again.");
    }
}
```

The search for these services is carried out by the `searchServices()` method of the agent. This method is used by passing a NULL attribute set, the UUID set, which identifies the service we are trying to locate, along with the remote device reference and Listener itself.

b) Checking for the OBEX server presence.

We first need to make sure the phone can handle the OBEX exchange via OBEX push. This step is taken care with the search for services.

You will have noticed that we initiated the search for services with a UUID set. This value, 0x1105, represents the OBEX object push profile service. If this service is present on the remote phone then a service record will be returned. Once we have the service record we can proceed to the next stage, that of obtaining an address to the remote phone. However, without the presence of this service we cannot begin to exchange objects between the two phones.

c) Obtain OBEX address of device

Once we have established the presence of a Bluetooth OBEX service on the remote phone, we need to establish its address. The service search will have returned a service record for the remote phone.

```
public void searchCompleted(ServiceRecord servRecord, String message)
{
    //cache the service record for future use
    this.servi ceRecord = servRecord;

    try
    {
        if(servi ceRecord==nul l)
        {
            setAl ert("Tel l a friend", "No servi ces were found on thi s bluetooth devi ce",
                Al ertType.ERROR);
        }
        el se
        {
            String url =
                servi ceRecord.getConnecti onURL(Servi ceRecord.NOAUTHENTI CATE_NOENCRYPT,
                    fal se);
            sendBtMessage(url );
        }

        // if the search was not cancelled, then di spl ay the resul ts
        if(i sSearchConti nued)
        {
            setMai nMenu();
        }
    }
    catch (Excepti on e)
    {
        setAl ert("Search Compl eted: "+e.toString());
    }
}
```

We do this by interrogating the `getConnecti onURL()` method from the phone's `servi ceRecord` object and passing it to the `MI DI et` method `sendBtMessage()`.

5.1.3 Connecting to the remote device

Now that we have selected the remote phone we wish to send an object to, established that it is capable of receiving an OBEX push and established its Bluetooth address, we are now ready to start the actual job of sending data to it.

This is the point where we start using the open source Aventana package.

This package provides an interface for sending the object from one phone to another. In section 4.2 we investigated how object exchange might be achieved in the absence of the OBEX packages from the Symbian OS 7.0s Java implementation. We could have sent the object by creating the bytes manually and then packaging them together with the correct header and sent them down the stream, as we have demonstrated in section 4.1. What the Aventana API does is provide us with a more usable interface for interacting with the Bluetooth stack.

In essence this package reflects the functionality of the JSR 82 OBEX API.

The other really useful part this package plays is that it takes the address of the OBEX push service on the remote phone and converts it to a BTSPS service URL, which is something we can use.

We use this new BTSPS address to communicate over the RFCOMM layer to send the OBEX object.

```
if (!url .startsWi th("btgoep: //"))
```

```
        throw new IOException ("Only OBEX Connections supported");
    url = "btspp" + url.substring(6);
```

This URL is then used to open the connection to the remote phone.

```
    ClientSession cs = (ClientSession) OBEXConnector.open (adr);
```

We then convert the payload to a byte array:

```
    byte text[] = VCARD.getBytes("iso-8859-1");
```

Next we create a set of headers that identify what we are sending to the remote device. We populate these headers so that they describe the data packets we wish to send. Each header is set as a name and a value pair, i.e. Header ID (HI), Header Value (HV). So, for example, NAME and <payload name> would describe the name of the object we are sending. The headers are initially created in the following way:

```
    HeaderSet hs = cs.connect(cs.createHeaderSet());
```

Once the empty header set has been created, we specify each name value pair in turn. In this instance we have specified the NAME of the vCard, along with another header called LENGTH, which is the estimated length of the payload as a whole, calculated from the length of the byte array.

```
    hs.setHeader (headerSet.NAME, "SymbianSample.vcf");
```

After setting the NAME header we, then specify the TYPE of object we are sending, its LENGTH and the number of items we are passing. The COUNT header indicates how many objects will be sent during the course of this connection.. These headers all help the remote device gracefully know when to close the connection and if to expect any more.

```
    ...
    hs.setHeader (HeaderSet.TYPE, "vcard");
    hs.setHeader(HeaderSet.LENGTH, new Long(text.length));
    hs.setHeader (HeaderSet.COUNT, new Long(1));
    // NB. final bit sent
    hs.setHeader(0x49, text);
    ...
```

You will notice the addition of the 0x49 hexadecimal value. This denotes the final bit is being sent to the server. This provides the server with the information required to complete the exchange of data. Some mobile phones will not be able to complete the exchange without this action being taken, despite the flushing of data and the closing of the connection after transmission.

5.1.4 Sending the data over the connection

Now that the connection has been fed the required headers describing the action about to occur, we can proceed with the sending of the data with a PUT.

```
    // perform the OBEX 'PUT' - sends the headers to the recipient device
    Operation po = cs.put(hs);

    po.close();
    cs.disconnect(null);
    cs.close();
```

The data exchange is completed with the closing and disconnecting of the Operation and ClientSession objects.

6 Conclusion

This paper has examined alternative solutions to the lack of the optional OBEX API in Symbian OS 7.0s phones. This provides developers with the ability to add OBEX functionality to their Java MIDP applications. This can be used to exchange contact data or indeed form the basis of a viral marketing campaign.

By exploring an alternative way of using the Bluetooth protocol stack, we are able to provide a versatile solution that can be used across a range of Bluetooth devices, whether they have the OBEX API installed or not. This is not only restricted to mobile phones, but also to PCs, servers and small handheld devices.

It means that more Symbian phones can handle OBEX from within MIDP 2.0 in a client/server environment as well as object exchange. It does not require remote devices to be running a MIDP 2.0 application and will therefore help promote an application by word of mouth for that reason. Contact information could be distributed from within the application and then used to promote an SMS short code that provides the provisioning of a WAP download location as well as a billing model for the purchase of an application. The vCard data could in theory also be updated remotely via an HTTP server if this information were to change.

If the onus of sending viral objects is put inside the application, it removes any psychological barrier or human errors in promoting that application. This provides a greater incentive to promote the application, as the data transfer is free for the user.

7 Tables

7.1 OBEX operation codes

Opcode	Command	Definition
0x80	Connect	choose your partner, negotiate capabilities
0x81	Disconnect	signal the end of the session
0x02 (0x82)	Put	send an object
0x03 (0x83)	Get	get an object
0x04 (0x84)	Reserved	reserved
0x85 *high bit always set	SetPath	modifies the current path on
0xFF *high bit always set	Abort	abort the current operation
0x06 to 0x0F	Reserved	not to be used w/out extension to this specification
0x10 to 0x1F	User definable	use as you please with peer application

7.2 OBEX response codes

OBEX response code	HTTP status code	Definition
0x00 to 0x0F	None	reserved
0x10 (0x90)	100	Continue
0x20 (0xA0)	200	OK, Success
0x21 (0xA1)	201	Created
0x22 (0xA2)	202	Accepted
0x23 (0xA3)	203	Non-Authoritative Information
0x24 (0xA4)	204	No Content
0x25 (0xA5)	205	Reset Content
0x26 (0xA6)	206	Partial Content
0x30 (0xB0)	300	Multiple Choices
0x31 (0xB1)	301	Moved Permanently
0x32 (0xB2)	302	Moved temporarily
0x33 (0xB3)	303	See Other
0x34 (0xB4)	304	Not modified
0x35 (0xB5)	305	Use Proxy
0x40 (0xC0)	400	Bad Request - server couldn't understand request

0x41 (0xC1)	401	Unauthorized
0x42 (0xC2)	402	Payment required
0x43 (0xC3)	403	Forbidden - operation is understood but refused
0x44 (0xC4)	404	Not Found
0x45 (0xC5)	405	Method not allowed
0x46 (0xC6)	406	Not Acceptable
0x47 (0xC7)	407	Proxy Authentication required
0x48 (0xC8)	408	Request Time Out
0x49 (0xC9)	409	Conflict
0x4A (0xCA)	410	Gone
0x4B (0xCB)	411	Length Required
0x4C (0xCC)	412	Precondition failed
0x4D (0xCD)	413	Requested entity too large
0x4E (0xCE)	414	Request URL too large
0x4F (0xCF)	415	Unsupported media type
0x50 (0xD0)	500	Internal Server Error
0x51 (0xD1)	501	Not Implemented
0x52 (0xD2)	502	Bad Gateway
0x53 (0xD3)	503	Service Unavailable
0x54 (0xD4)	504	Gateway Timeout
0x55 (0xD5)	505	HTTP version not supported
0x60 (0xE0)	- - -	Database Full
0x61 (0xE1)	- - -	Database Locked

7.3 Connect request format

Byte 0	Bytes 1 and 2	Byte 3	Byte 4	Bytes 5 and 6	Byte 7 to n
0x80	connect packet length	OBEX version number	flags	maximum OBEX packet length	optional headers

7.4 OBEX request format

Byte 0	Bytes 1 and 2	Byte 3 to n
opcode	packet length	Headers or request data

7.5 Server response format

Byte 0	Bytes 1 and 2	Byte 3 to n
response code	response length	Response data

7.6 A PUT OBEX example

Client Request:	bytes	Meaning
opcode	0x80	CONNECT , Final bit set
	0x0007	7 bytes is length of packet
	0x10	version 1.0 of OBEX
	0x00	no connect flags
	0x2000	8K max packet size
Server Response:		
response code	0xA0	SUCCESS , Final bit set
	0x0007	packet length of 7
	0x10	version 1.0 of OBEX
	0x00	no connect flags
	0x0800	2K max packet size
Client Request:		
opcode	0x02	PUT (Final bit not set)
	0x0422	1058 bytes is length of packet
	0x01	HI for Name
	0x0017	Length of Name header
	vCard.vcf	name of object, null terminated
	0xC3	HI for Length
	0x00006000	Length of object is 0x6000
	0x48	HI for Object Body chunk
	0x0403	Length of Body header
	0x.	1K bytes of body

Server Response:		
response code	0x90	CONTINUE , Final bit set
	0x0003	length of response packet
Client Request:		
opcode	0x02	PUT , Final bit not set
	0x0406	1030 bytes is length of packet
	0x48	HI for Object Body chunk
	0x0403	Length of Body header, 1K plus HI and header length
	0x.	next 1K bytes of body
Server Response:		
response code	0x90	CONTINUE , Final bit set
	0x0003	length of response packet
Repeat this process until the final 1k chunk is to be sent		
Client Request		
opcode	0x82	PUT , Final bit set
	0x0406	1030 bytes is length of packet
	0x49	HI for End-of-Body chunk
	0x0403	Length of header (1K) plus HI and header length
	0x.	next 1K bytes of body
Server Response:		
response code	0xA0	SUCCESS , Final bit sent
	0x0003	length of response packet

Author Profile

Alan Newman is the Managing Director of Brighton based application development agency, Sensible Development (<http://www.sensibledevelopment.com>), who develop mobile and web based applications.

An established expert in mobile and web application development, business graduate Alan has had responsibility for process automation within the NHS, the creation of financial management and trading systems for HSBC and application development at sports portal Sportal.com before he set up Sensible Development in 2000.



Since then he has consulted for key players in the mobile industry along with contributing to Symbian Press's "Programming Java 2 Micro Edition on Symbian OS" (Martin de Jode) published in 2004. With one eye on technology and business convergence, Sensible Development has always taken a modular approach to development of applications. This enables business to market their products across a range of platforms whilst repackaging their products to extend market reach through the use of white labelling.

[Back to Developer Library](#)

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.