FlexiPanel

Bluetooth™ Source Code For Remote Devices

Contents

Introduction

This document contains hint and tips for writing code for remote Bluetooth devices when connecting to FlexiPanel Bluetooth products. These examples are taken from our FlexiPanel Client software and represent extracts of working code. Not all of the required headers, libraries, *etc*, are discussed so you will need some working knowledge of the platform you are working on.

Editing during transcription into this document may have introduced errors, so please let us know if anything is not right. This code comes with no guarantees! We provide this information freely in order to provide a complete service to our customers. This code remains our copyright and while we freely permit its use to interface to our products, you must obtain our permission before using it in any other way.

Windows

Typically, windows Bluetooth drivers require you to connect manually to the serial port service from the Bluetooth manager before executing the code that opens the I/O file. Data may flow in either direction at any time, so it is best to implement I/O as Overlapped I/O. Here is file opening code we typically use:

```
// Global connection variables
OVERLAPPED gpWriteOverlapped ;
OVERLAPPED gpReadOverlapped ;
HANDLE g_hFile;

// Connect() connects to com port iComPort which might either be
// the outgoing or incoming COM port.  Return value is zero on success
int Connect( int iComPort )
{
// set up COM string
   TCHAR sCom[16];
#ifdef _UNICODE
          wsprintf( sCom, L"\\\\.\\COM%d", iComPort );
#else
   sprintf( sCom, "\\\\.\\COM%d", iComPort );
#endif

// connect to com port
   g_hFile = CreateFile (sCom,GENERIC_READ | GENERIC_WRITE,0,NULL,OPEN_EXISTING,
                         FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,NULL);

   if (g_hFile == INVALID_HANDLE_VALUE )
   {
   g_hFile[iCh] = NULL;
   return GetLastError();
   }

   // set for no timeout
   COMMTIMEOUTS  cto;
   cto.ReadIntervalTimeout = 0;
   cto.ReadTotalTimeoutConstant = 0;
```

```
   cto.ReadTotalTimeoutMultiplier = 0;
   cto.WriteTotalTimeoutConstant = 0;
   cto.WriteTotalTimeoutMultiplier = 0;
   SetCommTimeouts( g_hFile, &cto );

   // Purge any buffered rubbish
   PurgeComm( g_hFile[iCh], PURGE_TXABORT | PURGE_RXABORT | PURGE_TXCLEAR |
                                PURGE_RXCLEAR );
   ZeroMemory( &gpReadOverlapped[iCh], sizeof(*gpReadOverlapped) );
   ZeroMemory( &gpWriteOverlapped[iCh], sizeof(*gpWriteOverlapped) );

   // create a read thread
   HANDLE hRepeatReadThread = CreateThread (NULL, 0, (LPTHREAD_START_ROUTINE)
                RepeatReadThread, 0, 0, NULL);

   // I don't need handle, so delete it
   CloseHandle(hRepeatReadThread);
}
```

Note how a separate read thread is used to manage asynchronous I/O. If you are unfamiliar with multithreaded processing, be aware that there sharing data between threads requires care (usually global variables), and that you may want to use semaphores (i.e. global variables) to allow threads to indicate their states to each other, e.g. for asking the read thread to close down on disconnection. We typically implement this as follows:

```
// semaphore to tell the read thread to finish.  Set this to
// true in the main thread to order this thread to shut down
bool gbAbandon = false;
#define MAXLEN 256

unsigned int RepeatReadThread(PVOID pDummy)
{
   unsigned char szBuff[MAXLEN];
   ZeroMemory( &szBuff, MAXLEN );

   DWORD NumChar = 1;  // or number of characters you are expecting
   DWORD NumRead = 0;

   if (!ReadFile (g_hFile, szBuff , NumChar, &NumRead, gpReadOverlapped ) )
   {
   int err = GetLastError();
   if (GetLastError()==ERROR_IO_PENDING)
   {
          while !HasOverlappedIoCompleted( &gpReadOverlapped[iCh]) && !gbQuitPlease)
          {
          Sleep(10);
          }
   }
   else
   {
          return err;
   }

   // deal with character(s) read in here
   }
   while (gbAbandon == false);

   return 0;
}
```

Sending data is more straightforward and is typically implemented as follows:

```
// Send nBytes bytes in buffer pBytes, timeout after TimeOutMultiplier units of 10ms
int TransmitBytes(const char* pBytes, unsigned int nBytes, int TimeOutMultiplier )
{
   unsigned int TimeoutCount = TimeOutMultiplier*nBytes+10;

   // Send bytes
   DWORD nBytesSent = 0xFFFFFFFF;
   if (!WriteFile( g_hFile, pBytes, nBytes, &nBytesSent, &gpWriteOverlapped ))
   {
   int err = GetLastError();
```

```
   if ( err!=ERROR_IO_PENDING )
   {
          return err;
   }
   else
   {
          while (!HasOverlappedIoCompleted(&gpWriteOverlapped))
          {
          if (TimeOutMultiplier)
          {
                 TimeoutCount--;
                 if (!TimeoutCount)
                 {
   return -10;  // my timed out code
                 }
          }
          Sleep(10);
          }
   }
   }

   return 0;
}
```

Close the file as normal. This does not close the Bluetooth connection and you could reconnect if you wish:

```
CloseHandle( g_hFile );
```

WinCE 3.0

There are differences in how you establish a Bluetooth connection between WinCE 3.0 and WinCE *.NET*. *.NET* allows you to manage the connection from software using windows sockets, whereas with 3.0 you still need to make the connection first using the Bluetooth manager. Some *.NET* platforms, including some of the iPAQ series, still use older Bluetooth drivers so the *.NET* Bluetooth services do not work and you should treat these as Win CE 3.0 devices from the perspective of Bluetooth. SmartPhone, at least on the Orange SPV we tested, fully implemented the *.NET* Bluetooth services.

Overlapped I/O is implicit but you need to handle timeout differently for WinCE than for Windows. Here is file opening code we typically use:

```
HANDLE g_hFile = NULL;    // file handle of com port

int Connect( int iComPort  )
{
   // create com string
   CString sCom;
   sCom.Format( L"COM%d:", iComPort );

   // connect to com port
   g_hFile = CreateFile (sCom,GENERIC_READ | GENERIC_WRITE,0,NULL,OPEN_EXISTING,
                         FILE_ATTRIBUTE_NORMAL,NULL);

   if (g_hFile == INVALID_HANDLE_VALUE )
   {
   g_hFile = NULL;
   return GetLastError();
   }

   // set  timeout
   COMMTIMEOUTS  cto;
   cto.ReadIntervalTimeout = 10;
   cto.ReadTotalTimeoutConstant = 10;
   cto.ReadTotalTimeoutMultiplier = 10;
   cto.WriteTotalTimeoutConstant = 10;
   cto.WriteTotalTimeoutMultiplier = 10;
   SetCommTimeouts( g_hFile, &cto );
```

```
   // Start read thread
   HANDLE hReadThread = CreateThread (NULL, 0, (LPTHREAD_START_ROUTINE) ReadThread, 0,
                                        0, NULL);

   // don't need handle
   CloseHandle( hReadThread );

   return 0;
}
```

Again, a separate read thread is used to manage asynchronous I/O. If you are unfamiliar with multithreaded processing, be aware that there sharing data between threads requires care (usually global variables), and that you may want to use semaphores (i.e. global variables) to allow threads to indicate their states to each other, e.g. for asking the read thread to close down on disconnection. We typically implement this as follows:

```
bool gbAbandon = false;
#define MAXLEN 256

unsigned int ReadThread(PVOID pDummy)
{
   unsigned char szBuff[MAXLEN];
   ZeroMemory( &szBuff, MAXLEN );

   DWORD NumChar = 1;  // or number of characters you are expecting
   DWORD NumRead = 0;

          ZeroMemory( &hh, sizeof(hh) );

          while (1)
          {
   if ( !ReadFile(g_hFile, szBuff, NumChar, &NumRead, NULL ) )
   {
          int err = GetLastError();

          // if you call file close in main thread, ERROR_GEN_FAILURE is thrown here
          // and you can interpret is as an abandon-read-thread indtruction
          if (err == ERROR_GEN_FAILURE)
          {
                 return ERROR_INVALID_HANDLE;
          }
          if (err)
          {
                 return err;
          }
   }

   // deal with character(s) read in here
}

return 0;
}
```

Sending data is again more straightforward. Because it's not overlapped explicitly, `WriteFile` may return before transmitting all characters. We typically implement this as follows:

```
int       Send( const char* pBytes, unsigned long nBytes, int TimeOutMultiplier )
{
   DWORD BytesWritten = 0;
   DWORD BytesLeft = nBytes;
   DWORD TotalBytesWritten = 0;
   DWORD TimeoutCount = TimeOutMultiplier*nBytes+10;

   while (nBytes>TotalNumberOfBytesWritten)
   {
          if (!WriteFile( g_hFile, pBytes, BytesLeft, &BytesWritten, NULL ))
          {
                 int err = GetLastError();
                 if ( err )
                 {
                        return err;
                 }
```

```
            }
            BytesLeft -= BytesWritten;
            TotalBytesWritten += BytesWritten;
            if (nBytes>TotalBytesWritten)
            {
                    TimeoutCount--;
                    if (!TimeoutCount)
                    {
                            return -10;  // my timed out response code
                    }
                    Sleep(10);
            }
    }

    return 0;
}
```

Close the file as normal. This does not close the Bluetooth connection and you could reconnect if you wish:

```
CloseHandle( g_hFile );
```

.NET Bluetooth sockets

The .NET development platforms provide Bluetooth extensions to Windows sockets. These are by far the most elegant Bluetooth implementations because the user is not required to interact with any Bluetooth management software in order to make a connection.

The following sample code was developed for WinCE *.NET* but the code for Windows *.NET* platform appears to be very much the same. We define the class `BTLib` for searching for devices and establishing a socket connection. Use `BTLib::FindDevices` to scan for devices. Then use `BTLib::Connect` and `BTLib::Disconnect` to establish a connection socket with a remote device.

```
#include <windows.h>
#include <winsock2.h>
#include <ws2bth.h>
#include <bt_sdp.h>
#include <bthapi.h>
#include <bt_api.h>

#include "BTLib.h"

#define MAX_DEVICES  16
#define MYBUFFSIZE      16384

typedef struct {
    TCHAR szName[256];
    BT_ADDR btaddr;
} MYBTDEVICE, *PMYBTDEVICE;

class BTLib
{
public:
    BTLib();
    ~BTLib();
    int16 FindDevices ( int ScanBigSecs ) ;
    SOCKET Connect ( HWND hwnd, int Device ) ;
    void Disconnect (  ) ;

public:
    MYBTDEVICE btd[MAX_DEVICES];     // List of BT devices
    int nDevs;                       // Count of BT devices
    SOCKET m_soc;
};

BTLib::BTLib( void )
{
    nDevs = 0;
    m_soc = INVALID_SOCKET;
```

```
}

BTLib::~BTLib( void )
{
}

int16 BTLib::FindDevices ( int ScanBigSecs )
{
    // follows book by Boling
    DWORD dwFlags, dwLen;
    HANDLE hLookup;
    int i, rc;
    int nMax = MAX_DEVICES;

    // Create inquiry blob to limit time of search
    BTHNS_INQUIRYBLOB inqblob;
    memset (&inqblob, 0, sizeof (inqblob));
    inqblob.LAP = 0x9e8b33;  // Default GIAC
    inqblob.length = ScanBigSecs;      // ScanBigSecs * 1.28  seconds
    inqblob.num_responses = nMax;

    // Create blob to point to inquiry blob
    BLOB blob;
    blob.cbSize = sizeof (BTHNS_INQUIRYBLOB);
    blob.pBlobData = (PBYTE)&inqblob;

    // Init query
    WSAQUERYSET QuerySet;
    memset(&QuerySet,0,sizeof(WSAQUERYSET));
    QuerySet.dwSize      = sizeof(WSAQUERYSET);
    QuerySet.dwNameSpace = NS_BTH;
    QuerySet.lpBlob      = &blob;

    // Start query for devices
    rc = WSALookupServiceBegin (&QuerySet, LUP_CONTAINERS, &hLookup);
    if (rc) return rc;

    PBYTE pOut = (PBYTE)LocalAlloc (LPTR, MYBUFFSIZE);
    if (!pOut) return -1;
    WSAQUERYSET *pQueryResult = (WSAQUERYSET *)pOut;

    for (i = 0; i < nMax; i++) {
        dwLen = MYBUFFSIZE;
        dwFlags = LUP_RETURN_NAME | LUP_RETURN_ADDR;
        rc = WSALookupServiceNext (hLookup, dwFlags, &dwLen, pQueryResult);
        if (rc == SOCKET_ERROR) {
            rc = GetLastError();
            break;
        }
        // Copy device name
        lstrcpy (btd[i].szName, pQueryResult->lpszServiceInstanceName);
        // Copy bluetooth device address
        SOCKADDR_BTH *pbta;
        pbta = (SOCKADDR_BTH *)pQueryResult->lpcsaBuffer->RemoteAddr.lpSockaddr;
        btd[i].btaddr = pbta->btAddr;
    }
    if (rc == WSA_E_NO_MORE) rc = 0;
    nDevs = i;
    WSALookupServiceEnd (hLookup);
    LocalFree (pOut);
    return rc;
}


SOCKET BTLib::Connect ( HWND hwnd, int Device )
{
    if (m_soc!=INVALID_SOCKET)
    {
        return INVALID_SOCKET;
    }

    m_soc = socket( AF_BT, SOCK_STREAM, BTHPROTO_RFCOMM );
    if (m_soc==INVALID_SOCKET)
    {
```

```
            return INVALID_SOCKET;
        }

        SOCKADDR_BTH sab;
        memset (&sab, 0, sizeof(sab));
        sab.addressFamily = AF_BTH;
        sab.btAddr = btd[Device].btaddr;
        sab.serviceClassId.Data1 = 0x00001101;
        sab.serviceClassId.Data2 = 0x0000;
        sab.serviceClassId.Data3 = 0x1000;
        sab.serviceClassId.Data4[0] = 0x80;
        sab.serviceClassId.Data4[1] = 0x00;
        sab.serviceClassId.Data4[2] = 0x00;
        sab.serviceClassId.Data4[3] = 0x80;
        sab.serviceClassId.Data4[4] = 0x5F;
        sab.serviceClassId.Data4[5] = 0x9B;
        sab.serviceClassId.Data4[6] = 0x34;
        sab.serviceClassId.Data4[7] = 0xFB;
        if (0 != connect (m_soc, (struct sockaddr*)&sab, sizeof(sab)) )
        {
            closesocket( m_soc );
            return INVALID_SOCKET;
        }

        return m_soc;
    }

    void BTLib::Disconnect (  )
    {
        closesocket( m_soc );
        m_soc=INVALID_SOCKET;
    }
```

Once you have called `BTLib::Connect` to obtain a socket to a remote device, you can use the sockets send() and recv() functions as normal, e.g.

```
        if ( SOCKET_ERROR == send( gsoc, pBytes, nBytes, 0 ) )
        {
            // handle error
        }

        NumberOfBytesRead = recv( gsoc, ((char*)lpBuffer, NumberOfBytesToRead, 0);
        if (NumberOfBytesRead==SOCKET_ERROR)
        {
            // handle error
        }
```

Whether or not the `recv()` function is blocking (i.e. does not return before `NumberOfBytesToRead` bytes have been received) may be platform dependent. We put the call inside a while loop to force it to be blocking and then implement a separate read thread as shown in the Windows and Windows CE examples.

Java for JABWT compliant devices

Our Java/Bluetooth implementations use Rocosoft's Rococo Simulator. To use our sample code you'll need to license this from Rococo (www.rococosoft.com), and add the file "isim_midp.jar" to your classpath.

Opening a connection from J2ME code is possible through the use of the `BTConnection` class, and the `BTConnectionListener` interface. Since the device and discovery methods are asynchronous (non-blocking), the `BTConnectionListener` provides notification of errors or completion in the discovery process.

To implement the `BTConnectionListener` interface, the Java/Bluetooth connection class must implement the `BTConnectionListener` interface. This merely requires implementing the `btStatus (int ...)` method, which is called to provide notification of connection related events.

```
        public class MyBluetoothConnection implements BTConnectionListener
       {
```

```
 public void btStatus (int status)
{
      switch (status)
      {
            case kDeviceDiscoveryStart:
              // Device discovery has started.
              break ;
            case kDeviceDiscoveryCancel:
              // Device discovery has been cancelled by user
              break ;
            case kDeviceDiscoveryEnd:
              // Device discovery has finished.
              break ;

            case kServiceDiscoveryStart:
              // Service discovery has started.
              break ;
            case kServiceDiscoveryEnd:
              // Service discovery has finished.
              break ;
            case kServiceDiscoveryEndFailure:
              // Service discovery failed due to an unknown error.
              break ;
            case kServiceDiscoveryEndNoRecords:
              // Service discovery finished, but no such service was found.
              break ;
      }
 }
}
```

The `BTConnection` object can be created as follows:

```
BTConnection btConnect = new BTConnection (kSerialPortService, this);
```

The first parameter is the UUID of the service we're searching for on the remote device.

The second parameter is to the object that implements the `BTConnectionListener` interface, which will receive notification of connection related events.

Sample code for discovering devices is as follows:

```
// Starting device discovery
btConnect.startDeviceDiscovery() ;

// Once the BTConnectionListener has received notification of device discovery
// completion, you can query the results of the search.

// Retrieving the number of devices found
int deviceCount = btConnect.getNumberOfDevices() ;

// Querying the name of a device. If the device doesn't have a name, this returns
// the Bluetooth address in its stead.
int index = ... ;
String deviceName = btConnect.getDeviceName(index) ;

// Retrieving the remote device's object (RemoteDevice is a JABWT class)
int index = ... ;
RemoteDevice selectedDevice = btConnect.getDevice(index) ;
```

Once you have fetched the `RemoteDevice` object, you can start service discovery:

```
RemoteDevice selectedDevice = btConnect.getDevice(index) ;

btConnect.startServiceDiscovery(selectedDevice, 0) ;

// The first parameter is the RemoteDevice object on which you wish to search for
// services.  The second parameter is the service UUID. If this is 0, the
// BTConnection object will instead use the UUID you specified when you created it.

// Once the BTConnectionListener has received notification of service discovery
// completion, you can query the results of the search.
```

```
// Retrieving the number of services found
int numRemoteServices = btConnect.getNumberOfServices() ;

// Retrieving the remote service string (which is what we need to initiate a
// connection)
int index = ... ;
String remoteService = btConnect.getService(index) ;
```

A connection is opened is as follows:

```
// Connecting to a service on a remote device
String remoteService = btConnect.getService(index) ;

btConnect.connect(remoteService) ;

// This will cause an alert to be displayed for the user to authorize a connection.
// If the user rejects this, a SecurityException is thrown.

// Since we almost always will be searching for a single service, we also provide a
// convenience method to connect to the one service found (as an alternative to the
// above).
btConnect.connectToLastServiceFound() ;
```

The following functions provide read and write functions. Like Palm, JABWT is big-endian, i.e. multi-byte integers are stores largest byte first, so the integer read and write functions include a facility to convert to/from little-endian data for cross-platform consistency and conformance with the FlexiPanel protocol. These methods will throw an exception if the read/write fails.

```
public void readChars (char[] readChars, int offset, int size)
{
	// Note: This is now the only method that actually reads from the connection.
	try
	{
		reader.read(readChars, offset, size) ;
	}
	catch (IOException _)
	{
		// Error reporting here
		throw _ ;
	}
}

public String readString(int size, boolean unicode)
{
	char[] readChars = null ;

	try
	{
		if (unicode)
		{
			char[] tempChars = new char[size] ;
			readChars = new char[size/2] ;

			readChars(tempChars, size) ;
			for (int i = 0 ; i < (size / 2) ; i++)
			readChars[i] = tempChars[i*2] ;
		}
		else
		{
			readChars = new char[size] ;
			readChars(readChars, size) ;
		}
	}
	catch (IOException _)
	{
		// Error reporting here
		throw _ ;
	}

	return new String(readChars) ;
}
```

```
public int readInt()
{
	int	size = 4 ;
	char	intBytes[] = new char[size] ;
	int	value = 0 ;
	int	nextByte ;

	try
	{
		for (int i = 0 ; i < size ; i++)
		{
			readChars(intBytes, i, 1) ;

			nextByte = (int) intBytes[i] ;
			value |= nextByte << (8 * i) ;
		}
	}
	catch (IOException _)
	{
		// Error reporting here
		throw _ ;
	}
	return value ;
}

public short readShort()
{
	int	size = 2 ;
	char[] shortBytes = new char[size] ;
	short value = 0 ;
	short nextByte = 0 ;

	try
	{
		for (int i = 0 ; i < size ; i++)
		{
			readChars(shortBytes, i, 1) ;
			nextByte = (short) shortBytes[i] ;

			value |= nextByte << (8 * i) ;
		}

	}
	catch (IOException _)
	{
		// Error reporting here
		throw _ ;
	}

	return value ;
}

public void writeChars (char[] writeBytes)
{
	try
	{
		writer.write(writeBytes) ;
		writer.flush() ;
	}
	catch (IOException _)
	{
		// Error reporting here
		throw _ ;
	}
}

public void writeInt(int value)
{
	// Send a 32 bit integer -
	int	size = 4 ;
	char	intByte ;
	int	nextByte ;
	int	offset = 0 ;
```

```
		try
		{
			for (int i = 0 ; i < size ; i++)
			{
				offset = 8 * i ; // Little Endian?
				nextByte = value & (0xFF << offset) ;
				intByte = (char) (nextByte >> offset) ;

				writer.write(intByte) ;
				writer.flush() ;
			}
		}
		catch (IOException _)
		{
			// Error reporting here
			throw _ ;
		}
	}

	public void writeShort(short value)
	{
		// Send a 16 bit integer
		int	size = 2 ;
		char	intByte ;
		int	nextByte ;
		int	offset = 0 ;

		try
		{
			for (int i = 0 ; i < size ; i++)
			{
				offset = 8 * i; // Little endian?
				nextByte = value & (0xFF << offset) ;
				intByte = (char) (nextByte >> offset) ;

				writer.write(intByte) ;
			}
			writer.flush() ;
		}
		catch (IOException _)
		{
			// Error reporting here..
			throw _ ;
		}
	}
```

Examples of calling the read / write functions:

```
	//Read a 32bit value
	int valueLong ;
	valueLong = btConnect.readInt() ;

	// Read a 16bit value
	short valueShort ;
	valueShort = btConnect.readShort() ;

	// Read in a variable sized block (this is the only method that actually reads from
	// the connection, all other read methods call this one.
	char[] buffer = new char[buffersize] ;
	btConnect.readChars(buffer, buffersize) ;

	// Or, to fill the buffer starting from an offset
	btConnect.readChars(buffer, offset, readSize) ;

	// Reading in a string
	boolean isUnicode = false ;
	String readString = btConnect.readString(size, isUnicode) ;

	// Writing a 32bit value
	int valueLong ;
	btConnect.writeInt(valueLong) ;

	// Writing a 16bit value
```

```
   short valueShort ;
   btConnect.writeShort(valueShort) ;

   // Writing a variable sized block (this is the only method that actually writes to
   // the connection, all other write methods call this one.
   char[] buffer = ... ; // Some character buffer filled with data
   btConnect.writeChars(buffer) ;
```

Closing a connection:

```
btConnect.close() ;
```

Palm OS

The following C++ object provides open, read, write and close functions for Bluetooth on Palm. Palm is big-endian, i.e. multi-byte integers are stores largest byte first, so the integer read and write functions include a facility to convert to/from little-endian data for cross-platform consistency and conformance with the FlexiPanel protocol.

```
#include "BtLib.h"

#define Swap16(x) _BtLibNetSwap16(x)
#define Swap32(x) _BtLibNetSwap32(x)

Err BtConnection::Connect()
{
   Err err = errNone ;
   SrmOpenConfigType config ;
   BtVdOpenParams btParams ;

   if (mLive)
          return err ; // Nothing to do here..

   // Build the connection configuration
   config.function = 0 ;
   config.drvrDataP = (MemPtr)&btParams) ;
   config.drvrDataSize = sizeof(BtVdOpenParams) ;

   // Build the UUID list
   const BtLibSdpUuidType kRfCommUUID == {btLibUuidSize16, {0x00, 0x03}} ;
   BtVdUuidList uuidList = {1, &kRfCommUUID} ;

   // Connect as client
   btParams.role = btVdClient ;

   // Blank out the address (to ensure the discovery dialog is presented)
   for (int i = 0 ; i < 8 ; i++)
          btParams.u.client.remoteDevAddr.address[i] = 0 ;

   btParams.u.client.method = btVdUseUuidList ;
   btParams.u.client.u.uuidList = uuidList ;

   // Open the serial connection
   err = SrmExtOpen(sysFileCVirtRfComm, &config, sizeof(config), &mPortID) ;
   mLive = (err == errNone)? true : false ;

   return err ;
}


Err BtConnection::Disconnect()
{
   Err err = errNone ;

   if (mLive)
   {
          // Flush the buffers
          SrmReceiveFlush(mPortID, 0) ;
```

```
            SrmSendFlush(mPortID) ;

            err = SrmClose(mPortID) ;
            mLive = false ;
      }

      return err ;
   }

   // The code for writing to the connection (which all the write methods will call)
   bool BtConnection::Write(UInt8* data, UInt32 size)
   {
      Err            err ;
      UInt16  index = 0 ;

      SrmSendFlush(mPortID) ;
      SrmReceiveFlush(mPortID, 0) ;

      while (index++ <= NUM_RETRIES)
      {
            // *** NOT FINAL: needs to be fixed. Will fail if NUM_RETRIES > 0
            // and the first send fails or is incomplete
            if (SrmSend(mPortID, (void*) data, size, &err) == size)
                  break ;
      }

      return (err == errNone) ? true : false ;
   }

   bool BtConnection::WriteInt (UInt32 number, bool swap)
   {
      UInt32 writeNum ;

      if (swap)
            writeNumber = Swap32(writeNum) ;
      Err err = Write ((UInt8*) &writeNum, sizeof(writeNum)) ;
      return (err == errNone) ? true : false ;
   }

   bool BtConnection::WriteShort(UInt16 number, bool swap)
   {
      UInt16 writeNum ;

      if (swap)
            writeNum = Swap16(writeNum) ;
      Err err = Write ((UInt8*) &writeNum, sizeof(writeNum)) ;
      return (err == errNone) ? true : false ;
   }


   bool BtConnection::ReadInt(UInt32* number, bool swap)
   {
      UInt32  readNum = 0 ;
      bool           didRead = false ;

      didRead = Read ((UInt8*) &readNum, sizeof(readNum)) ;
      *number = (swap) ? Swap32(readNum) : readNum ;
      return didRead ;
   }

   bool BtConnection::ReadShort(UInt16* number, bool swap)
   {
      UInt16  readNum = 0 ;
      bool           didRead = false ;

      didRead = Read ((UInt8*) &readNum, sizeof(readNum)) ;
      *number = (swap) ? Swap16 (readNum) : readNum ;
      return didRead ;
   }

   bool BtConnection::Read (UInt8* data, UInt32 size)
   {
      Err            err = errNone ;
      UInt16  index = 0 ;
```

```
    UInt32  bytesRead = 0 ;
    bool            didRead = false ;

    while (index++ <= NUM_RETRIES)
    {
            // *** NOT FINAL: needs to be fixed. Will fail if NUM_RETRIES > 0
            // and the first read fails or is incomplete
            bytesRead = SrmReceive(mPortID, data, size, SysTicksPerSecond()
                                    * READ_TIMEOUT, &err) ;
            if (bytesRead == size)
            {
                    didRead = true ;
                    break ;
            }
    }

    return didRead ;
}
```