Dialogic®
Making Innovation Thrive™

Application Note

Using Dialogic® APIs in .NET Applications

Using Dialogic® APIs in .NET Applications

Application Note

Executive Summary

This application note discusses a way of using Dialogic® APIs (referred to in this application note as Dialogic® R4 API and Dialogic® Global Call API) in .NET applications that are written in managed high-level .NET programming languages, such as Visual Basic.NET (VB.NET) or Visual C#®. This application note examines the issues involved and proposes a solution containing a working example that includes a mixed mode Visual C++® class library, which interacts with Global Call API and R4 API on one side and with the VB.NET application on the other.

Sample code, available for download, provides a simple VB.NET application interacting with Dialogic® Host Media Processing Software Release 3.0 for Windows® to make/receive SIP calls and perform some simple media operations, like play/record files, send/receive DTMF digits, and transaction recording.

Using Dialogic® APIs in .NET Applications

Application Note

Table of Contents

Using Dialogic® APIs in .NET Applications

Application Note

Introduction

The Microsoft® .NET Framework introduced a new advanced platform for application developers. One of the major features in this platform is the Common Language Runtime (CLR), allowing multiple projects written in different .NET languages to be combined and developed as a single solution in a common Integrated Development Environment (IDE).

To benefit from CLR, however, the developer writes managed code such that class objects and other variables of complex types are the subject of a new memory management procedure (like garbage collector and others), unlike "native" programs where an object always had the same address as long as it remained in execution scope.

Because pre-.NET C or Visual C++® code cannot be compiled as a CLR module, and hence cannot call any other .NET solution component directly, a developer who wants to benefit from CLR using pre-existing Visual C++ code (to avoid massive re-writing or learning new languages) must create a managed extension to the "native" code.

This application note provides a simple implementation of a managed extension to existing native Visual C++ classes, which allows previously written Dialogic-enhanced Visual C++ classes (for Dialogic® APIs [referred to in this application note as Dialogic® Global Call API and Dialogic® R4 API]) to be consumed by applications written in .NET languages, like Visual Basic.NET (VB.NET) or Visual C#®.

This application note contains an example that includes a mixed mode Visual C++ class library, which interacts with Global Call API and R4 API on one side and with the VB.NET application on the other, and downloadable sample code (see the *For More Information* section) of a simple VB.NET application interacting with Dialogic® Host Media Processing Software Release 3.0 for Windows® to make/receive SIP calls and perform some simple media operations, like play/record files, send/receive DTMF digits, and transaction recording. The version of the sample code in this application note was created using Visual Studio® .NET 2003.

Problem Definition

For this example, a collection of Visual C++ native (unmanaged) classes are compiled as a DLL and linked to the C Run-Time (CRT) libraries including Dialogic® CRT libraries, and these classes are to be used in a managed Visual Basic® (or Visual C#) application without re-writing the existing code. The object then is to consume as much as possible of the existing functionality in VB.NET. It is also desirable to be able to communicate between Visual C++ and Visual Basic in two directions, by calling Visual C++ methods from Visual Basic for commands, and calling Visual Basic from Visual C++ for events and return values.

Issues when Consuming Native Code from Managed Visual Basic® Applications

The following are examples of the attempts and the issues that may result when trying to make Visual C++ code usable in the managed Visual Basic workspace:

- Export the Visual C++ classes using the dllexport attribute (just like would be done with native Visual C++ consumer).

  Issue: It most likely will not work because VB.NET, by its nature, cannot import and use native classes as is; it can reference managed objects and classes only.

- Turn the Visual C++ code into a collection of methods and export each method as a DLL function.

  Issues:

  - It may work with some serious limitations, but it will require a major rewrite of the existing code and a great amount of coding on the Visual Basic level, where the classes will be recreated using those imported methods.

  - For complex data types, like structures, arrays, or strings, the managed code uses managed types (String, Array, IntPtr, etc.), which cannot be passed directly to the imported native methods, so a quite complex data marshalling mechanism has to be implemented when passing arguments from Visual Basic to the imported functions.

- It may be difficult to make some methods work. For example, to play a file, the DX_IOTT structure is re-defined in the Dialogic-enhanced Visual Basic code (assuming one knows how to marshal the fields). Then, the play function is kicked off, passing this structure and others as arguments. The R4 API and Global Call API require that the IOTT and other structured arguments remain in scope during an entire playback. What happens next is that even though Dialogic-enhanced Visual Basic IOTT structure is in scope during the playback, it becomes unreachable in the Visual C++ method after a while. This is because an object on the managed heap may be replaced at any time by the garbage collector, and Dialogic-enhanced Visual C++ code knows nothing about a new position of the IOTT. There are some methods in .NET that prevent data from being moved by the garbage collector (GCRoot), but excessive use of such techniques may compromise the .NET memory management.

Issues when Calling Managed Code from Unmanaged Library

Dialogic® libraries use asynchronous operations and provide information, as an event, about a component's state transition, such as incoming call offered, playback completed, etc. A notification mechanism needs to be created to inform the Dialogic-based Visual Basic application when such an event occurs on Dialogic® objects. One of the ways to do so is to create a special callback function in Visual Basic class and pass its address to the unmanaged Visual C++ code, so the Visual C++ code will call this function each time when some event is detected. These callback functions are called "delegates" in .NET programming.

Issue: The delegate function is a managed object, and its location (or an address of the entry point) can be moved at any time. Moving it will cause the Visual C++ code to use an invalid pointer each time when the managed heap reallocation occurs. So, a pointer to a managed function cannot just be passed to unmanaged code and expected to work well. One of the possible solutions for this is discussed in the *Proposed Solution* section.

Proposed Solution

To help resolve the issues described above and to make good use of existing code, an intermediate component (or Visual C++ class) can be created, which acts as an interface to both the managed Visual Basic application and unmanaged Visual C++ code. Such code is called "mixed mode code." Mixed mode components link to CRT libraries, and may contain both managed and unmanaged data types and objects. Sometimes .NET developers call mixed-mode classes "wrapper classes," because such classes "wrap" a managed interface around unmanaged functionality, translate calls, and marshal data between these two. So, instead of re-writing the existing C or Visual C++ code, it is possible to "wrap" it using a relatively small and simple class.

High Level Design

Figure 1 shows the main components of the proposed solution, and the relationships between them. As discussed previously, the Visual Basic application must be able to access native methods and receive event notification from the Visual C++ library via a callback function.

The solution consists of three major components:

1. **Unmanaged or CRT** module defines native Visual C++ classes (like CVoiceDev or CGCDev) and provides an entry/exit point to the .NET application (defined in VoiceDev.h/cpp, GCDev.h/cpp, GCBoard.h/cpp).

2. **Mixed mode code** contains a managed class that can be used directly in a managed application. Since this code is linked to CRT, the private (non-exportable) members of this class may use unmanaged objects (defined in DlgcLibNet.h/cpp and built in DlgcLibNet.dll).

3. **Managed or .NET application** (Visual Basic is used in this application note). This application includes a reference to the exportable (managed) part of the mixed mode code, and thus can use its classes as if the classes were defined in the Visual Basic application.

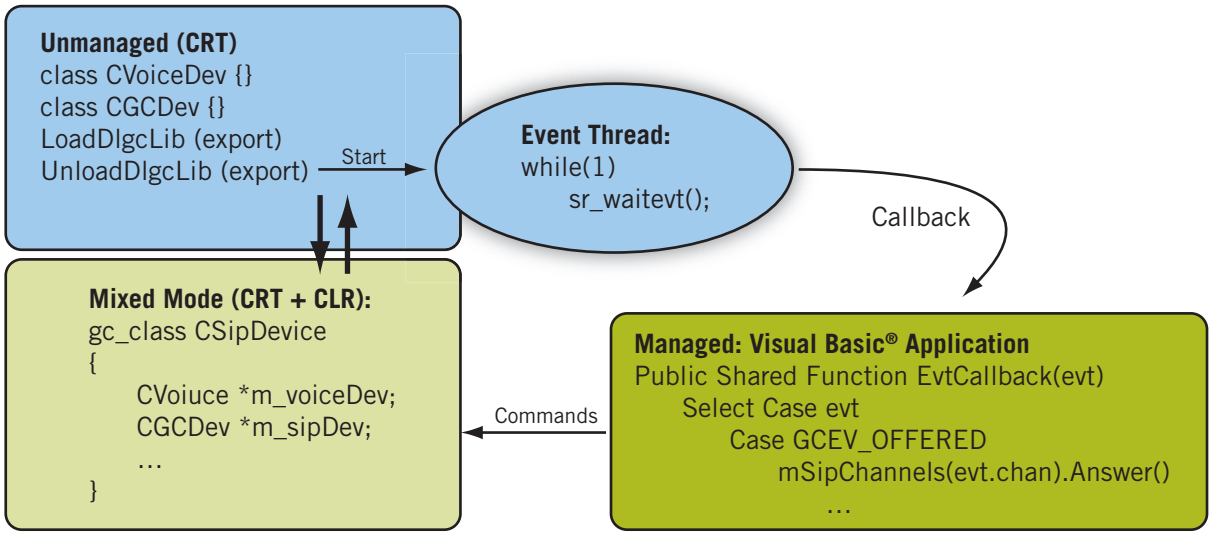

Figure 1. High Level Design for Proposed Mixed Mode Code Solution

The Visual Basic application defines a delegate that is used as a callback function for Dialogic® events that will be passed from the mixed code. Upon startup, the Visual Basic application calls a static exportable function from DlgcLibNet and passes the address to this delegate. The DlgcLibNet starts a thread and waits for Dialogic events in this thread infinitely. Once an event is received, the library calls the delegate passing the event information as an argument, so the Visual Basic application can process this event.

Class Description

Unmanaged Classes

CVoiceDev: Base class CDevice

Declared: VoiceDev.h, **Defined:** VoiceDev.cpp

This class provides basic voice functionality such as record, playback, send/receive digits, and transaction recording. It automatically selects an audio encoding for play/record according to a file extension.

Public functions:

- Open() — Opens and initializes a voice resource
- Close() — Closes the resource
- Reset() — Stops I/O activity on the device and resets it to the idle state
- Play() — Plays a file
- Record() — Records audio to a file
- transRecord() — Records audio from two different timeslots (sources) to a file
- Dial() — Plays one or more DTMF tones
- GetDigits() — Waits for a specific amount of DTMF during a specific time
- DigBuffer() — Returns a string containing collected digits

- Listen() — Connects the resource to the CT Bus timeslot
- Unlisten() — Disconnects the resource from the CT Bus timeslot
- CloseFile() — Closes a file handle used for play/record. Consumer must call this function upon TDX_PLAY or TDX _RECORD to avoid a handle leak.

CGCDev: Base class CDevice

Declared: GCDev.h, **Defined:** GCDev.cpp

This class controls SIP and media (Real-time Transport Protocol [RTP]) resources and allows making or receiving SIP calls using different audio coders.

Main public functions include:

- Open() — Opens and initializes SIP and media (RTP) devices
- Close() — Closes the resource
- Reset() — Resets the resource to the idle state
- MakeCall() — Places an outbound SIP call
- Accept() — Sends 180 RINGING to inbound SIP call
- Answer() — Answers inbound SIP call
- Drop() — Disconnects existing SIP call; is ignored if no call is present
- Release() — Releases resources occupied by a SIP call and returns the resource to the idle state
- Listen() — Connects the media resource to the CT Bus timeslot
- Unlisten() — Disconnects the resource from the CT Bus timeslot

Managed Classes

The following two managed classes are defined in namespace DlgcLibNet:

- public_gc class CSipDevice
- private_gc class CEvtThread

public_gc class CSipDevice

Declared: DlgcLibNet.h, **Defined:** DlgcLibNet.cpp

This class uses instances of CVoiceDevice and CGCDevice to provide the combined functionality and create a simple logical "channel" for VoIP calls using SIP protocol. This class can be used in other .NET applications. This class contains pointers to unmanaged CVoiceDev and CGCDev objects. Its public functions are serving as a managed wrapper for the unmanaged classes, and include the following functions:

- OpenSipDev()
- CloseSipDev()
- MakeCall()

- AnswerCall()
- DropCall()
- ReleaseCall()
- Reset()
- PlayFile()
- RecordFile()
- WaitDigits()
- StopVoice()

It also contains the following properties:

- TerminateOnDigit
- ChannelNumber
- DevText
- ChannelState

private_gc class CEvtThread

This class is private in the namespace and therefore is not exposed to the consumer application. This class is a helper class, and its function is to control a thread where Dialogic events are captured and passed to the consumer's delegate.

The **DlgcLibNet** namespace also contains two control functions:

- **LoadDlgcLib** — Used to initialize static variables needed for library's functionality
- **UnloadDlgcLib** — Called at the end of operations to gracefully stop the event thread and clean up used resources

These two functions could be included into CSipDevice as static members. The "standalone" mode was chosen to illustrate another way of exporting functionality from a mixed mode library.

Consumer Application Overview

In the given example, the consumer application is written in VB.NET, but C# or Visual C++.NET could be other options.

The Visual Basic application includes the DlgcLibNet.dll as a reference, so it can use any publicly managed object from this namespace directly, without re-defining anything.

The application constructs an array of CSipDevice classes, creates objects, opens "channels," and defines a delegate as a place where Dialogic events are passed. It maintains some simple state machine and works with GUI controls to implement the example's functionality. Of course, the application can be more complex and useful, depending how well the wrapper class is elaborated.

Brief Description of the Consumer Application

The example consists of two Visual Basic modules:

- **DlgcDefinitions.vb** — Re-defines Dialogic events that may be used in this application and declares a few constants
- **FormMain.vb** — Defines the state machine and user interface

The application imports two control functions, LoadDlgcLib and UnloadDlgcLib, as described in the *Managed Classes* section, using the *Declare* statement:

```
Declare Function LoadDlgcLib Lib "DlgcLibNet" ( _
  ByVal x As Integer, ByVal x As CallBack) As Integer
  Declare Function UnloadDlgcLib Lib "DlgcLibNet" () As Integer
```

It also includes the DlgcLibNet.dll in the reference list to be able to access the publicly managed CSipDevice class defined in this DLL.

The application creates a global template array where objects of CSipDevice will be stored:

```
Public Shared mSipChannels As Array
```

The application also needs to declare a callback function prototype, which is somewhat similar to *#typedef* in C. The statement below defines a function type called "CallBack" that receives three Int32 arguments and returns a Boolean value:

```
Public Delegate Function CallBack(ByVal evtChannel As Integer, _
 ByVal evtType As Integer, ByVal evtParam As Integer) As Boolean
```

And then lastly, the application must declare, create, and define the callback function itself and pass its address to the LoadGlgcLib() function, so the library will be able to call this function each time an event occurs on one of Dialogic® devices, thus passing the channel number, event type, and (optionally) data associated with the event. This is done in three steps as follows:

1. Define the callback function itself:

```
Public Shared Function EvtCallback(ByVal evtChannel As Integer, ByVal evtType As
Integer, ByVal evtData As Integer) _
 As Boolean
   Dim strTxt As String
   Dim evtIdx As Integer
   Select Case evtType
     Case enGcEvents.GCEV _ TASKFAIL To
…
  End Select
   Return True
  End Function
```

Note that the callback function in step 1 has the same arguments and return type as the "CallBack" type previously described for the callback function prototype.

2. However, the address of the callback function cannot be passed to the DlgcLibNet because this function is a part of a managed frmMain class and therefore is a subject of the garbage collector. To "pin" this address, copy the callback function to a static memory (VS 7) or use its handle (VS2005/2008). Declare a global static variable of CallBack type:

```
Private Shared cbCallBack As CallBack, constract it by copying a context of the
EvtCallback: cbCallBack = New CallBack(AddressOf EvtCallback),
```

And only then, pass the cbCallBack function pointer to the DlgcLibNet.

This pointer will not be moved by the garbage collector, so the static library can safely call this function at any time:

```
rc = LoadDlgcLib(mintNumOfChannels, cbCallBack)
```

3. Lastly, contract an array of the SIP channels by creating each instance, opening the channel, and pushing it into the array:

```
mSipChannels = Array.CreateInstance(GetType(CSipDevice), mintNumOfChannels)
  Dim tmpSipDev As CSipDevice
  For I As Int32 = 1 To mintNumOfChannels
    tmpSipDev = New CSipDevice
    mSipChannels.SetValue(tmpSipDev, I - 1) 'adding it to the array
    If (tmpSipDev.OpenSipDev(I) <> 0) Then 'opening device
      Exit For
    End If
  Next I
```

After these steps are done, assign some functionality to the GUI controls and work out a state machine.

Building the Mixed Mode Library

When a developer creates a new class library project, Visual Studio® automatically sets properties and attributes appropriate for the managed library only. Such a library, by default, will not link to CRT libraries and therefore cannot use CRT libraries in the code. To enable a mixed mode in a new class library, set the following options:

- Add msvcrt.lib to the link list (Property->Linker->Input)
- Exclude nonchkclt.dll from the list (that is, remove it from Additional Dependencies property on the same page)
- Force the code to be linked to CRT by adding __DllMainCRTStartup@12 to the Force Symbol References property on the same page.

Visual Studio creates a class library with no entry point. This means that DLLMain will not be executed when a process or a thread loads the library. Because static native libraries are used and some static variables have to be initialized (like a CallBack routine address), some entry point is needed to the library. For this purpose, the example defines a “manual” entry point, LoadDlgcLib() function, and has the consumer call this function at startup.

For More Information

A Zip file containing the sample code can be downloaded at http://www.dialogic.com/goto/?11176

Managed Extensions for Visual C++® Programming —
http://msdn2.microsoft.com/en-us/library/aa712574(vs.71).aspx

Dialogic® Media Processing Boards (Dialogic® DM3 Media Boards and Dialogic® JCT Media Boards) —
http://www.dialogic.com/products/tdm_boards/media_processing/default.htm

Dialogic® Host Media Processing (HMP) Software —
http://www.dialogic.com/products/ip_enabled/hmp_software.htm

Dialogic Manuals — http://www.dialogic.com/manuals/default.htm

Dialogic®
Making Innovation Thrive™

www.dialogic.com

Dialogic Corporation
9800 Cavendish Blvd., 5th floor
Montreal, Quebec
CANADA H4M 2V9

10/08 11070-01