**CICS Transaction Gateway for Windows V7.0**

# CICS TG: Developing .NET components for CICS connectivity

*Version Date:* February 21, 2008

Paul Crockett
IBM Hursley
44-1962-815304
paul.crockett@uk.ibm.com

Licensed Materials - Property of IBM

© Copyright IBM Corporation 2007, 2008
All Rights Reserved

US Government Users Restricted Rights – Use, duplication, or disclosure restricted by GSA ADP Schedule Contract with IBM Corporation

# Overview and Background

The CICS Transaction Gateway provides connectivity through a variety of APIs - Java, C++ and C to name a few. Although there is currently no support for running in a .NET memory managed environment, it is possible to use a mixed mode DLL to bridge the gap between .NET and the CICS Transaction Gateway. This article demonstrates how to use such an approach to exploit the CICS Transaction Gateway from a .NET environment. A working knowledge of C++ and either C# or Java is assumed. Visual Studio 2003 or higher is required to build the accompanying source code.

A .NET managed environment (such as ASP.NET on IIS) is similar to a Java Virtual Machine in that memory is allocated and freed automatically, alleviating the developer of the responsibility. When executing unmanaged code from a managed environment it is therefore necessary to use an intermediate library which is responsible for marshalling data and function calls between the managed and native environments. These libraries are known as mixed mode DLLs due to the fact that they contain both managed and unmanaged code. The relationship between mixed mode libraries and managed and unmanaged code is shown in figure 1.

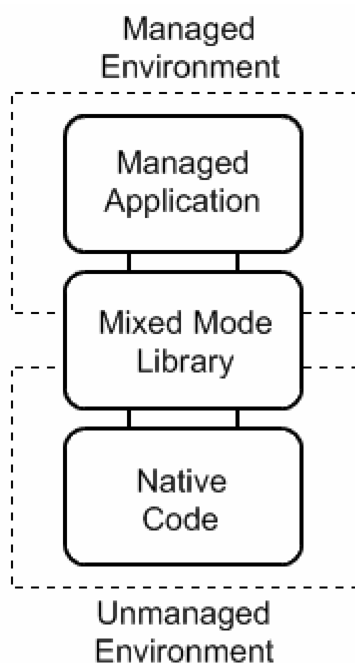


Figure 1: Relationships between mixed mode libraries and managed and unmanaged code

Mixed mode DLLs typically expose managed classes to .NET which are responsible for invoking the native functions they encapsulate. This article documents the creation of a simple mixed mode DLL wrapper around the ECI components of the CICS Transaction Gateway API for C which will allow .NET applications to communicate with CICS using the CICS Transaction Gateway.

© Copyright IBM Corporation 2007, 2008

# Creating a mixed mode Visual Studio project

Visual Studio 2002 added support for Managed C++, a variant of C++ with added keywords and constructs for the creation of managed code. Managed C++ has since been superseded by C++/CLI, introduced with Visual Studio 2005. C++/CLI uses a syntax that is neater than Managed C++ and more similar to that of C#. If working on a new product using Visual Studio 2005 or 2008, C++/CLI is recommended. A full list of the differences between the two languages can be found at:

http://msdn2.microsoft.com/en-us/library/ms235289(VS.80).aspx

The first step towards creating a mixed mode DLL is to create a new project within Visual Studio. The project type for managed and mixed mode libraries is the Visual C++ Class Library. This project type is automatically configured to create mixed mode libraries with either Managed C++ or C++/CLI.

The code snippets shown in this tutorial are C++/CLI. The accompanying zip file contains the C++/CLI code as well as the Managed C++ equivalent.

**Note:**
When compiling Managed C++ from within Visual Studio 2005 or later, you must change the Common Language Runtime support option to Old Syntax (/clr:oldSyntax), otherwise C++/CLI is assumed and the code will not compile. This setting can be found under **Configuration Properties → General**.

To set up a Visual Studio project to use the CICS Transaction Gateway C API the following steps must be performed:

1. The Additional Include Directories project setting must be changed so that it contains the *\include* subdirectory of the CICS Transaction Gateway installation, for example *C:\Program Files\IBM\CICS Transaction Gateway\include*. This setting can be found under **Configuration Properties → C++ → General**.

2. The Additional Library Directories project setting must be changed so that it contains the *\lib* subdirectory of the CICS Transaction Gateway installation, for example *C:\Program Files\IBM\CICS Transaction Gateway\lib*. This setting can be found under **Configuration Properties → Linker → General**.

3. The Additional Dependencies project setting must be changed so that it contains the cclwin32.lib library. This setting can be found under **Configuration Properties → Linker → Input**.

© Copyright IBM Corporation 2007, 2008

## Creating a managed class

All managed classes must exist within a namespace. Namespaces in .NET are roughly equivalent to packages in Java. Each .NET class can be specified using a fully qualified name which includes the containing namespace, or namespaces can be imported to aid readability, thus removing the need to fully qualify each class. Importing a namespace in .NET is equivalent to importing a package in Java. In Managed C++ and C++/CLI, namespaces are imported with the `using namespace` statement, as shown in figure 2.

```
#define CICS_W32
#include "cics_eci.h"

using namespace System;
using namespace System::Runtime::InteropServices;
```

Figure 2: Importing namespaces (CicsEci.h)

The sample class is called `CicsEci` and is placed within the `EciWrapper` namespace. The `CicsEci` class will encapsulate the `CICS_ExternalCall` function call, which takes a parameter of type `ECI_PARMS`. Therefore the `CicsEci` class will have an unmanaged pointer to an `ECI_PARMS` structure. It will also hold a reference to a managed byte array used as a commarea, and a delegate which can be used as a callback function pointer. These fields are shown in figure 3.

```
namespace EciWrapper {
    public ref class CicsEci
    {
    private:
        //Fields
        ECI_PARMS*        eciParms;
        array<Byte>^      commareaArray;
        EciCallback^      callbackDelegate;
    public:
        //Constructor and destructor
        CicsEci();
        ~CicsEci();
    };
}
```

Figure 3: Fields in the CicsEci class (CicsEci.h)

The `ref` keyword in the class definition specifies that the `CicsEci` class is a managed class and that references to any instances of it should be managed by the .NET framework. The ^ (caret or hat) symbol denotes a handle to a managed class.

The `commareaArray` field is of type `array<Byte>^` which is a managed pointer to a managed array of type `System::Byte`. This means that responsibility for managing any memory associated with the array lies with the .NET framework. It also means that whenever the array is accessed, an implicit check is performed and an `IndexOutOfRangeException` is thrown if an attempt is made to access an index outside the bounds of the array. The `Byte` data type is equivalent to the C++ `char` type and the two are interchangeable. In this case `Byte` has been used to avoid ambiguity, since the `char` type in C++ and the `char` type in C# are of different sizes.

The `EciCallback` data type is defined later in this article.

## Constructors and destructors

As with other OOP languages, .NET classes may contain constructors and destructors. The `CicsEci` class allocates and initializes the unmanaged `ECI_PARMS` block in the constructor and deletes it in the destructor. The constructor and destructor are shown in figure 4.

```
CicsEci::CicsEci() {
    //Create the ECI_PARMS structure
    this->eciParms = new ECI_PARMS;
    memset(this->eciParms, 0, sizeof(ECI_PARMS));

    //Set ECI version
    this->eciParms->eci_version = ECI_VERSION_1A;

    //Set callback to null
    this->callbackDelegate = nullptr;
}
CicsEci::~CicsEci() {
    //Delete ECI_PARMS
    delete this->eciParms;

    //Set callback to null
    this->callbackDelegate = nullptr;
}
```

Figure 4: CicsEci constructor and destructor (CicsEci.cpp)

There two types of destructor in C++/CLI. The standard destructor, which is prefixed by the ~ (tilde) symbol, compiles into a `Dispose` method and the containing class is changed so that it implements the `System::IDisposable` interface. The `IDisposable.Dispose` method can be called from user code when the object is no longer needed, so this type of destructor is often referred to as a deterministic finalizer.

The second type of destructor is prefixed by the ! (exclamation) symbol and compiles into a non-deterministic finalizer which is called when the garbage collector is about to delete the object from memory. It is recommended that this type of destructor is only used when a deterministic destructor is also specified. The `CicsEci` class does not use this type of destructor.

© Copyright IBM Corporation 2007, 2008

## Properties

.NET properties are special method pairs which can be used like fields. Each property may define a get and a set method which will be called implicitly when the property is accessed in code. The `CicsEci` class exposes many members of the `ECI_PARMS` structure as properties. The declarations of three of the properties are shown in figure 5 as an example.

```
property String^ AbendCode {
    String^ get();
}
property array<Byte>^ Commarea {
    array<Byte>^ get();
    void set(array<Byte>^ commarea);
}
property short Timeout {
    short get();
    void set(short timeout);
}
```

Figure 5: Declaration of the AbendCode, Commarea and Timeout properties (CicsEci.h)

Since `AbendCode` will be a read-only property, no `set_AbendCode` method is declared or implemented. The implementation of these three properties is shown in figure 6.

```
String^ CicsEci::AbendCode::get() {
    return CicsEci::NtvCharToMgdString(this->eciParms->eci_abend_code, 4);
}
array<Byte>^ CicsEci::Commarea::get() {
    return this->commareaArray;
}
void CicsEci::Commarea::set(array<Byte>^ commarea) {
    this->commareaArray = commarea;
}
short CicsEci::Timeout::get() {
    return this->eciParms->eci_timeout;
}
void CicsEci::Timeout::set(short timeout) {
    this->eciParms->eci_timeout = timeout;
}
```

Figure 6: Implementation of the AbendCode, Commarea and Timeout properties (CicsEci.cpp)

The `NtvCharToMgdString` method is defined in the next section.

© Copyright IBM Corporation 2007, 2008

## Strings

The internal representation of the `System::String` class is an array of `System::Char`, equivalent to `wchar_t` in C++. However, the `ECI_PARMS` structure uses fixed length `char` arrays, so some care must be taken when converting between the two.

The `CicsEci` class defines the `MgdStringToNtvChar` method for copying the first N characters of a managed `String` object into an unmanaged `char` array, using the `System::InteropServices::Marshal` class to perform the conversion.

The `NtvCharToMgdString` method performs the reverse operation, converting characters in an unmanaged `char` array into a managed `String` object. This operation is much simpler since the `String` class provides a constructor which takes a pointer to a `char` buffer and a length.

The implementation of both functions is shown in figure 7.

```
void CicsEci::MgdStringToNtvChar(String^ src, cics_char_t* dest, int len) {
    cics_char_t*   ntvChars;

    //Zero destination block
    memset(dest, 0, len);

    if (src != nullptr) {
        //Get unmanaged char array
        ntvChars = (cics_char_t*)
Marshal::StringToHGlobalAnsi(src).ToPointer();

        //Copy to destination pointer
        if (src->Length > len) {
            memcpy(dest, ntvChars, len);
        } else {
            memcpy(dest, ntvChars, src->Length);
        }

        //Free temp memory
        Marshal::FreeHGlobal((IntPtr) ntvChars);
    }
}
String^ CicsEci::NtvCharToMgdString(cics_char_t* src, int len) {
    String^ mgdString;

    //Measure the string up to a null character or the maximum length len
    int realLen = 0;

    while ((realLen < len) && (src[realLen] != 0)) {
        realLen++;
    }

    //Create managed string and trim off any trailing whitespace
    mgdString = gcnew String(src, 0, realLen);
    return mgdString->TrimEnd(' ');
}
```

Figure 7: MgdStringToNtvChar and NtvCharToMgdString functions (CicsEci.cpp)

© Copyright IBM Corporation 2007, 2008

## Delegates

Native callback functions can be wrapped by .NET delegates. Delegates are type-safe function pointers which can point to instance as well as static methods. The `CicsEci` class stores the callback function as a delegate and converts it to a native function pointer before invoking `CICS_ExternalCall`. The `EciCallback` delegate type is declared using the `delegate` keyword. Since this is a type definition, `EciCallback` can be used like any other reference type. The definition of the type is shown in figure 8.

```
[UnmanagedFunctionPointer(CallingConvention::Cdecl)]
public delegate void EciCallback(short returnCode);
```

Figure 8: EciCallback delegate (CicsEci.h)

The `UnmanagedFunctionPointer` attribute specifies that the delegate should be treated as a `cdecl` function when used in unmanaged code. If no `UnmanagedFunctionPointer` attribute is applied, the default calling convention is `stdcall`.

**Note:**
The `UnmanagedFunctionPointer` attribute is important if passing a delegate to native code. If an incorrect calling convention is specified, the application using the library may crash.

The get and set methods for the `Callback` property can now use this delegate type as shown in figure 9.

```
EciCallback^ CicsEci::Callback::get() {
    return this->callbackDelegate;
}
void CicsEci::Callback::set(EciCallback^ callback) {
    this->callbackDelegate = callback;
}
```

Figure 9: Callback property get and set methods (CicsEci.cpp)

**Note:**
Extra care must be taken when using the asynchronous callback mechanism to ensure that the object associated with the delegate is not garbage collected before the asynchronous call completes, as this could lead to unexpected behaviour.

© Copyright IBM Corporation 2007, 2008

# Enumerations

The eci_calltype and eci_extend_mode members of the ECI_PARMS structure should be set to one of the predefined ECI constants. In .NET it is possible to declare these values as constants within the CicsEci class. However, the convention in .NET is to group related constants into special value types called enumerations. Therefore the ECI constants used with the ECI_PARMS block are grouped into two enumerations within the EciWrapper namespace. The definitions of the EciCallType and EciExtendMode enumerations are shown in figure 10.

```
public enum class EciCallType : short
{
    Sync =                     516, /* Synchronous call.                */
    Async =                    517, /* Async call used with ECI_GET_REPLY. */
    AsyncNotifyMsg =           518, /* Async call, notify by message.   */
    AsyncNotifySem =           519, /* Async call, notify by semaphore. */
    GetReply =                 520, /* Used to get reply to ASYNC call. */
    GetReplyWait =             521, /* As above but wait for reply.     */
    GetSpecificReply =         528, /* Get specific ASYNC reply.        */
    GetSpecificReplyWait =     529, /* As above but wait for reply.     */
    StateSync =                522, /* Synchronous request for CICS status. */
    StateAsync =               523, /* As above but async, no notify.   */
    StateAsyncSem =            524, /* As above but notify by semaphore. */
    StateAsyncMsg =            525, /* As above but notify by message.  */
};
public enum class EciExtendMode : short
{
    NoExtend =                 0,   /* These values are to be used with  */
    Extended =                 1,   /* call_types other that ECI_GET_xxx & */
    Commit =                   2,   /* ECI_STATE_xxx.                   */
    Cancel =                   Commit,
    Backout =                  3,   /*                                  */
    StateImmediate =           4,   /* All ECI_STATE_xxx call-types.    */
    StateChanged =             5,   /* ECI_STATE_ASYNC_SEM &            */
    StateCancel =              6,   /* ECI_STATE_ASYNC_MSG call_types only. */
};
```

Figure 10: EciCallType and EciExtendMode enumerations (CicsEci.h)

The CicsEci properties which encapsulate the eci_calltype and eci_extend_mode members can now use these enumerations as regular integral types as shown in figure 11.

```
EciCallType CicsEci::CallType::get() {
    return (EciCallType) this->eciParms->eci_call_type;
}
void CicsEci::CallType::set(EciCallType callType) {
    this->eciParms->eci_call_type = (cics_sshort_t) callType;
}
EciExtendMode CicsEci::ExtendMode::get() {
    return (EciExtendMode) this->eciParms->eci_extend_mode;
}
void CicsEci::ExtendMode::set(EciExtendMode extendMode) {
    this->eciParms->eci_extend_mode = (short) extendMode;
}
```

Figure 11: CallType and ExtendMode properties (CicsEci.cpp)

The return codes for the CICS_ExternalCall and CICS_EciListSystems native calls are grouped into the EciReturnCodes enumeration in the same way.

## Handles

In a managed environment, native handles and pointers are represented by the `System::IntPtr` type. Since `IntPtr` is a value type, it can be directly converted to and from native pointer types. The `eci_async_notify` member of the `ECI_PARMS` structure can contain a handle to either a window or an event. This member is encapsulated by the `Handle` property of the `CicsEci` class, as shown in figure 12.

```
IntPtr CicsEci::Handle::get() {
    return IntPtr(this->eciParms->eci_async_notify.sem_handle);
}
void CicsEci::Handle::set(IntPtr handle) {
    this->eciParms->eci_async_notify.sem_handle = (cics_lhandle_t)
handle.ToPointer();
}
```

Figure 12: Handle property (CicsEci.cpp)

**Note:**
Although the `ECI_PARMS` member is called `sem_handle`, the `Handle` property should be set to the handle of an event object, such as `System::Threading::AutoResetEvent` or `System::Threading::ManualResetEvent`, and not a `System::Threading::Semaphore`.

## The CICS_ExternalCall native function

In the CicsEci class, the majority of the work involved with placing values in and reading values from the ECI_PARMS structure is performed by the properties. The only members of ECI_PARMS which may need to be set before the call are eci_commarea, eci_commarea_length and eci_callback.

The pin_ptr type can be used to hold a pinned reference to an object. When an object reference is assigned to the pointer, the object is pinned by the garbage collector, meaning it cannot be deleted or moved in memory until it has been unpinned. Therefore the pointer can safely be passed to unmanaged functions which may modify the memory it points to. The Execute method uses a pinned pointer to hold a reference to the commarea memory during the native call.

The Marshal class provides the GetFunctionPointerForDelegate method which takes a delegate as a parameter and returns an IntPtr that can be used to invoke the delegate from a native environment. The Execute method uses this to set the eci_callback member.

After the call to CICS_ExternalCall, the method tests the return code. If the return code is a value other than ECI_NO_ERROR, an EciException is thrown. Unlike Java, .NET methods do not need to declare which exception types they throw.

The implementation of the Execute method is shown in figure 13.

```
void CicsEci::Execute() {
    short           ret;
    pin_ptr<Byte>   commareaPtr;

    //Pin the managed commarea and assign to ECI_PARMS
    if (this->commareaArray != nullptr) {
        commareaPtr = &this->commareaArray[0];
        this->eciParms->eci_commarea = commareaPtr;
        this->eciParms->eci_commarea_length = this->commareaArray->Length;
    }

    //Set up the callback if needed
    if (this->callbackDelegate != nullptr) {
        this->eciParms->eci_callback = (CICS_EciNotify_t)
Marshal::GetFunctionPointerForDelegate(this->callbackDelegate).ToPointer();
    }

    //Make the CICS call
    ret = CICS_ExternalCall(this->eciParms);

    //Check the return code
    if (ret != ECI_NO_ERROR) {
        //Throw an exception
        throw gcnew EciException(ret, this->AbendCode);
    }
}
```

Figure 13: Execute method implementation (CicsEci.cpp)

The sample CicsEci class also defines a ListSystems method which encapsulates the CICS_EciListSystems native function call.

© Copyright IBM Corporation 2007, 2008

## Using the CicsEci class

If the mixed mode DLL compiles successfully, it can be used from C# or VB.NET by adding it as a reference. Once referenced, each managed type exported by the DLL can be used.

The EciB1 sample program is provided in both C# and VB.NET and demonstrates the `CicsEci` class by querying the client daemon for a list of defined CICS servers and then launching an ECI request on the chosen server. Both samples can be found in the accompanying zip file.

The sample program requires a CICS server which returns the commarea as ASCII text. However, other client applications may handle text in the EBCDIC encoding. The `System::Text::Encoding` class provides access to all codepages installed on the client machine, so ASCII can be substituted for EBCDIC as shown in figure 14.

```
//Display commarea as EBCDIC (C#)
Encoding encEbcdic = Encoding.GetEncoding("IBM037");
Console.Write("EBCDIC text: ");
Console.WriteLine(encEbcdic.GetString(eciObj.Commarea));

'Display commarea as EBCDIC (VB.NET)
Dim encEbcdic As Encoding = Encoding.GetEncoding("IBM037")
Console.Write("EBCDIC text: ")
Console.WriteLine(encEbcdic.GetString(eciObj.Commarea))
```

Figure 14: Using EBCDIC character encoding

This tutorial has demonstrated how to create a mixed mode DLL to wrap the CICS External Call Interface API of the CICS Transaction Gateway. The same process can be applied to create managed wrappers for other native code.

© Copyright IBM Corporation 2007, 2008