Native Functions

by
Tom "Blitz" Conder
Carlos "c0mpi1e" Cuello

Dated: 9/13/1999

Introduction

The Unreal engine has the ability to mix Unrealscript (UScript) and C++ code. In order to call C++ functions from UScript you must compile them into a dynamic-link library (DLL) file. Calling these functions, known as native functions, is probably one of most overlooked features of Unreal. This document will discuss implementing native functions, identify a few gotchas and give a step-by-step guide to a basic example.

Consider the Benefits

By far one of the most impressive features of Unreal is its highly modularized and replaceable game code. Amateurs can modify the game by replacing or augmenting its code, sounds, meshes and levels with their own. These modifications called 'mods' are the concern of an entire online community. Mods keep Unreal fresh and interesting and give it life long after most have solved the original. Native functions are one of the tools in the toolchest of mod authors. They allow for algorithmic textures, performance-critical artificial intelligence, access to the Win32 API and implementation hiding.

Be Wary of the Thorns

There are disadvantages to using native functions. First of all, the process is complicated and not well documented. Although this document will shed some light on process, writing them is unsupported by Epic Games. Secondly, forget Mac compatibility. Writing DLLs means that your mod will only work in Microsoft Windows. Lastly, every patch will break your mod. UScript is more portable between patches, particularly beta patches. Also, C++ headers and library files are needed to compile DLLs. That means you need the latest public source to compile your DLL.

Set up the Package

Mod authors should start by setting up a new package. A package file is a file that contains a collection of related objects, eg. textures, sounds and scripts. Package files containing UScript classes are identified with a '.u' extension. Refer to the documentation entitled 'Package Files' on the Unreal Tech web site (http://unreal.epicgames.com/) for more information about package files.

Unreal comes with its own compiler capable of producing package files. It is called 'ucc.exe' and is found in the 'Unreal\System' directory. We will discuss using the compiler later.

Start your mod by making a directory under the Unreal folder with the name of your project. For example, if your project is named 'Hazard', then create a directory named 'Unreal\Hazard'.

Next, create four directories under your package folder named Classes, Inc, Src and Lib. The Classes directory will hold your UScript. The Inc directory will hold the public header files. The Src directory will hold your C++ files and private header files. The Lib directory will hold static library file created during the compilation.

In order for your new package to be seen by the Unreal Editor, the ucc compiler and during gameplay you need to edit the Unreal.ini file. It is located in the 'Unreal\System' directory. Find the [Editor.EditorEngine] section and add 'Hazard' to the [EditPackages] list, eg. EditPackages[?]=Hazard.

We discussed how to set up a new package. We took a look at the Unreal compiler, created the needed directories and edited the appropriate initialization file. Now that the system is ready, it is time to use the Unreal compiler to automatically generate the public header file.

Generate the Public Header File

Classes that use native functions must be declared as native in UScript. Unreal always looks for C++ implementation of native function in a DLL corresponding to the class's package name. For example, if your package is named 'Hazard', Unreal will look for a file named 'Hazard.dll' in the System directory.

The following example implements a class named 'hzTest'. It is a subclass of another class named Actor. Notice we declare your class as native.

```
class hzTest extends Actor
    native;
```

The Unreal compiler, ucc, will read the class file and automatically generate a header file which you can include in your DLL project. Be careful not to edit this file as the Unreal compiler will see that it has changed and offer to overwrite it. Although Epic's documentation recommends running the ucc compiler as an external tool through Visual C++, you do not want to do it for this step. *Run the ucc compiler from the DOS prompt*, eg. 'ucc make'. Since the compiler prompts you for a response and the prompt does not show up in VC++ output window, it appears in fact to hang the C++ compiler.

Running ucc from the DOS prompt is easy. Simply, go to the DOS prompt. Find the Unreal\System directory. Now type: 'ucc make'. Eventually the resulting prompt will appear:

```
The file '..\Hazard\Inc\HazardClasses.h' needs to be updated. Do you want to overwrite the existing version? (Y/N)
```

When prompted, press the 'Y' key to answer in the affirmative.

We generated out public header file. This file exposes functions declarations and definitions to other classes. Lets move on implementing the private header file.

Make the Private Header File

A private header file contains local definitions and structures that we do not want to expose to other classes. The private header file must include the engine, core and public header files. The following file, HazardPrivate?.h, is an example of a private header file.

```
// ========================================================================
// Project: Native Function Document
//
// Description:
// This is the header file for the native function document DLL.
// ========================================================================

#include "Engine.h"
#include "Core.h"

// The HazardClasses header file is automatically generated by the ucc
// compiler.
#include "HazardClasses.h"
```

We wrote our private header file. Any private structure or definitions go here. Now we are to implement the C++ source file.

Hack the C++ Source Code

Write your C++ file. Be sure to include your private header file. Take a look at the following example, a file named hzTest.cpp.

```
// ========================================================================
// Project: Native Function Document
//
// Description:
// This file contains code for the native function document DLL.
// ========================================================================

#include "HazardPrivate.h"

IMPLEMENT_PACKAGE(Hazard);

IMPLEMENT_CLASS(AhzTest);

IMPLEMENT_FUNCTION(AhzTest, 1700, execIncTest);

// execIncTest - called as IncTest method in UScript
void
AhzTest::execIncTest (FFrame& Stack, RESULT_DECL)
{
    // input parameter handling
    guard(AhzTest::execIncTest);
    P_FINISH;
```

```
    GLog->Logf(TEXT("in IncTest() "));

    iTest++;

    // return the result
    *(DWORD*)Result = iTest;

    unguardexec;
}
```

This example simply increments the value of a variable named iTest by one and returns the resulting value. The next section explains the macros used in this example and describes a few more.

Mind the Macros

When implementing native functions in C++ you need to learn some of the macros found in the source code. The above code example uses a few of them: IMPLEMENT_PACKAGE, IMPLEMENT_CLASS, IMPLEMENT_FUNCTION and P_FINISH, guard(), unguard.

IMPLEMENT_PACKAGE() - takes as a parameter the name of the package; takes the parameter in the form IMPLEMENT_PACKAGE (MyPackage?);

IMPLEMENT_CLASS() - takes as a parameter the name of the class in the form: IMPLEMENT_CLASS(MyClassName?);

IMPLEMENT_FUNCTION() - declaration macro to expose the function to the source file; takes three parameters in the form:

```
IMPLEMENT_FUNCTION( AClassName, UniqueID, AFunctionName);
```

where AClassName is the name of your class, =UniqueID=* is an integer that is to uniquely identify the function. This number must be unique to any package loaded in Unreal. For example you can have multiple functions with the same name in your package, they could be in different classes, the numbers are what tells the Unreal which function to call. Unreal uses several numbers for its functions throughout the Engine/Unreali/UnrealShare/Core/Fire etc package files. However, most of those stay below approximately 500, and depending on which packages are loaded by Unreal, you may use those, too. Personally, I try and keep all of mine above 1000.

P_GET_UBOOL, **P_GET_STRUCT**, **P_GET_INT**, etc. - These macros grab the variable off of the call stack.

Let us take the following example with a native function declared in UScript and C++:

```
native function FunctionName( int x, int y, bool bDo ); // UScript
void execFunctionName( FFrame &amp;Stack, void* Result ); // C++
```

Now those two declarations look **very** different. How do you get those parameters? Well, the answer lies in the use of the Stack and the P_ macros. The stack frame contains the current function states, including the parameters. FFrame &Stack is a reference to the current stack frame and state. The P_GET* macros grab the parameters off of the stack. You **must** use the P_GET* calls to grab the parameter types in the same order as is defined in UScript.

In our example, we would want to do the following

```
void MyClass::execFunctionName( FFrame &Stack, void const *Result )
{
    guard( MyClass::execFunctionName );

    P_GET_INT( x );
    P_GET_INT( y );
    P_GET_UBOOL( bDo );

    P_FINISH;


    // Do what we want.

unguard;
}
```

The P_GET* macros all perform the same function functions although they might look a little different depending on the type. For a complete listing of all of the macros, look at Unreal\Core\Inc\UnScript.h

P_GET_UBOOL_OPTX, **P_GET_INT_OPTX**, etc. - used in the same way as the regular P_GET macros, but add on an extra parameter, the default value. For example, following the previous code listing:

```
native function Foo( optional int x = 10, optional int y = 10 );
```

```
void MyClass::execFoo( FFrame &Stack, void const *Result )
{
     guard( MyClass::execFoo );
     P_GET_INT_OPTX( x, 10 );
     P_GET_INT_OPTX( x, 10 );
     P_FINISH;

     // Do what we want.


     if ( x == 10 )
          GLog->Log( TEXT( "Using defaults" ) );
     unguard;
}
```

P_GET_ARRAY_REF, **P_GET_STR_REF**, etc. - used in the exact same way as the P_GET* macros (**not** the optional macros), and allows parameters to be paseed by reference. For example:

```
native function Foo( out int x, out int y );
void MyClass::execFoo( FFrame &Stack, void const *Result )
{
     guard( MyClass::execFoo );
     P_GET_INT_REF( x );
     P_GET_INT_REF( x );
     P_FINISH;

     // Do what we want.

     unguard;
}
```

P_FINISH - indicates the end of the call stack. You **must** put this in your native functions or Unreal **will** crash by not closing up the stack and trying to load a new function, state or class.

guard(), **unguard** - creates an exception handler for the following block of code. Basically, it translates to a try...catch block, and adds a name to unwinding stack. Here is the usage:

```
guard(MyClass::MyFunction)

or

guard(CoolBlockOfCode)</p>
```

If an exception is raised in the block of code wrapped by a guard, then the log file will show the relevant stack info, including the deepest stack name, eg.:

```
void MyFunc()
{
    guard(MyClass::MyFunc);

    // Do something to raise an exception

    unguard;
}
```

This code will cause the log file to give not only the relevant error, but also where in the code it happened. You can add as many guard ()'s in a function as you want, the more you do it, the more specific an error will be.

I once started a function that had about 20 lines of code. An error was being thrown somewhere. I started narrowing it down, adding guard()s and unguard statements everywhere. Finally I narrowed it down to the specific line.

Return the Result

Native functions can return values. Unreal defined a variable named 'Result'. To return a value from a function set the value of the variable equal to the returned value. Since the returned value it cast to its desired type it is possible to return just about anything from a function including integers, float and references to user-defined structures.

Now that we have taken a look at C++ source code, let us move on to the final step: compiling the source code into a DLL file. Unfortunately, in order to get your code to compile successfully you will need to customize the compiler setting. We will take a look at

that in the next section.

Tweak the C++ Compiler Settings

The Visual C++ compiler settings must be tweaked in order for the DLL to be compiled. First of all the preprocessor settings must be changed. In the Preprocessor category under the C/C++ compiler options tab, remove the existing preprocessor setting and replaced them with: WIN32,WINDOWS,_WINDOWS,UNICODE,_UNICODE,HAZARD_API=__declspec(dllexport)

These settings tell the compiler to use the Win32 API, Unicode and to give the HAZARD_API a definition. The HAZARD_API defition is important become the generated header file has a bug where it sets HAZARD_API to DLL_IMPORT. In actually we want the defintion to indicate dll exports, so defining it here works around the problem without editing the automatically generated header file.

Also, in the preprocessor tab set the additional include directories to point to the core, engine and package include directories. That is, set it to: ..\..\Core\Inc,..\..\Engine\Inc,..\Inc

Secondly, the link settings must be changed. In the General category under the Link options tab, set the output file name to: ..\..\System/Hazard.dll. Additionally, set the Object/library modules to point to the core and engine library files. That is, set it to: ..\..\Core\Lib\Core.lib ..\..\Engine\Lib\Engine.lib