

[Prerequisites](#)

[Development](#)

[Implement the XSplit script plugin interface](#)

[Export the required functions](#)

[Explicitly name exported functions using a module-definition \(DEF\) file](#)

[Specify C linkage in exported functions](#)

[Optional: Verify a DLL's exported function names](#)

[Testing a custom DLL in XSplit Broadcaster](#)

Prerequisites

- Win32 C++ compiler
- Alternatively, **we highly recommend using Visual Studio** which includes the MSVC compiler and also provides a complete development environment for developing Windows applications and libraries.

Note: The code samples in this document assume that you are developing in Visual Studio 2010 or newer. Some preprocessor macros used in these samples rely on the Active Template Library (ATL), whose header files are bundled with Visual Studio.

- Windows SDK for Windows 7 or newer

The latest SDK release can be found [here](#). The Windows SDK is also bundled with recent versions of [Visual Studio](#).

| Visual Studio version | Bundled Windows SDK version |
|---------------------------------|-----------------------------|
| Visual Studio 2015 | Windows 10 SDK |
| Visual Studio 2013 | Windows 8.1 SDK |
| Visual Studio 2012 | Windows 8.0 SDK |
| Visual Studio 2010 Professional | Windows 7 SDK |

Development

Implement the XSplit script plugin interface

The following functions¹ must be implemented in the global namespace:

```
BOOL WINAPI XSplitScriptPluginInit();
BOOL WINAPI XSplitScriptPluginCall(IXSplitScriptDllContext * pContext, BSTR
functionName, BSTR * argumentsArray, UINT argumentsCount, BSTR * returnValue);
void WINAPI XSplitScriptPluginDestroy();
```

You must define *IXSplitScriptDllContext* as follows:

```
[uuid("9A554D8A-912F-4F1E-9A26-CC002B3B99BD")]
DECLARE_INTERFACE_(IXSplitScriptDllContext, IUnknown)
{
    STDMETHOD(Callback)(BSTR functionName, BSTR* argumentsArray, UINT
argumentsCount) PURE;
};
```

Keep in mind that the minimum supported Windows OS version by XSplit Broadcaster is Windows 7 SP1. Ensure that you either properly handle scenarios in which you might call Win32 API functions that are only supported in newer OS versions, or avoid calling them at all.

The sample Visual Studio project is set up with everything mentioned in the following sections to get you started developing custom DLLs for XSplit Broadcaster. The project is compatible with Visual Studio 2010 or newer.

If you choose to implement the functions above by modifying the sample project, you may choose to skip over the succeeding sections and continue on to [this section](#).

XSplitScriptPluginInit is called when your DLL is loaded in XSplit; typically, this is done during XSplit's initialization (when the splash screen is visible).

If, for some reason, you determine that your DLL cannot be run, you may return FALSE to indicate an initialization error and let XSplit unload your DLL. Otherwise, return TRUE.

¹ WINAPI is a Win32 preprocessor macro specifying that a function should use the stdcall calling convention (equivalent to the `__stdcall` keyword).

It is possible to execute your initialization code here, but it is recommended that you do so in *XSplitScriptPluginCall* instead, when a source or extension calls any of your methods.

XSplitScriptPluginDestroy is called when your DLL is about to be unloaded. You may run your cleanup or finalization code here.

XSplitScriptPluginCall is the entry point of a source or extension into your DLL. This is where you must respond to requests made by the source or extension.

functionName the name of the method that the source or extension wants to call.
argumentsArray is an array of string arguments passed with the call, while ***argumentsCount*** is the number of arguments passed (or the length of ***argumentsArray***).

When a source or extension makes a method call, XSplit goes through each loaded DLL one by one until one of them handles the call. If the DLL handles the call, *XSplitScriptPluginCall* must return *TRUE* to notify XSplit and stop the search. If *FALSE* is returned, XSplit calls the next DLL's *XSplitScriptPluginCall* method, and so on, until one of them handles it.

If another DLL happens to handle a function name that your DLL has also handled, only one of those DLLs may receive the call. Therefore, it is highly recommended to introduce some uniqueness to function names in order to avoid name clashes. For example, instead of expecting a *functionName* of "MyTestFunction", you could prefix the DLL's name along with it, making it "MyDllName.MyTestFunction".

You can pass a string back to the caller through the ***returnValue*** pointer. To use this, you must allocate a BSTR string by passing an OLECHAR string to the Win32 function *SysAllocString* and assign the returned BSTR to address pointed at by *returnValue*.

pContext is a pointer to a context associated with the calling instance of the source or extension. This context allows you to execute callbacks on that particular instance. It can be stored for future use, which is useful for sending event notifications that extend beyond the method scope of *XSplitScriptPluginCall*, or as extra return mechanism in addition to the *returnValue* pointer.

Export the required functions

To make the implemented functions accessible to XSplit, they must be exported by the DLL. The only requirement is that the exported function names must be as declared above -- that is, without name decoration or name mangling. However, C++ names are decorated by default using compiler-specific naming conventions. There are two² ways to avoid this:

² There is a third way, which is to use the `#pragma comment(linker, "/EXPORT:"xxx)` convention, but there are a lot of steps involved and it is specific to MSVC; the two methods above are faster and simpler to do.

- Define a module-definition (DEF) file which explicitly declares the exported function names to be used by the linker (recommended).
- In the source code, declare the functions to be exported with C linkage (extern "C"). Note that this will not work in MSVC.³

Explicitly name exported functions using a module-definition (DEF) file

You can configure the linker to use a DEF file which explicitly names the exported functions. The DEF file should be defined⁴ like this:

```
EXPORTS
    XSplitScriptPluginInit      PRIVATE
    XSplitScriptPluginCall      PRIVATE
    XSplitScriptPluginDestroy   PRIVATE
```

In MSVC, if a DEF file is not specified, the exported names will contain a "leading underscore (_)" and a trailing at sign (@) followed by the number of bytes in the parameter list in decimal" [as described here](#).

Specify C linkage in exported functions

Depending on your compiler/linker, it may also be required to specify in the functions' declarations that they have C linkage (see the **extern "C"** keyword or the equivalent **EXTERN_C** macro), instead of C++ linkage, in order to prevent C++-style name decoration:

```
extern "C"
{
    BOOL WINAPI XSplitScriptPluginInit();
    BOOL WINAPI XSplitScriptPluginCall(IXSplitScriptDllContext * pContext, BSTR
functionName, BSTR * parametersArray, UINT parametersCount, BSTR * returnValue);
    void WINAPI XSplitScriptPluginDestroy();
}
```




This is not required in MSVC, however; simply configuring the linker to use a DEF file as described above is sufficient to export the proper names with C linkage. You may need to take other steps to prevent name decoration, depending on your compiler; refer to the compiler's documentation for more information.

³ In MSVC, exported functions declared with `__stdcall` / `WINAPI`, such as the above mentioned ones, will have a leading underscore added to their names even if they have C linkage.




⁴ `PRIVATE` specifies that the function name should not be included in the import library; even if this is specified, `XSplit` will be able to access these functions as long as they are exported.



Optional: Verify a DLL's exported function names

If you want to verify that the exported names in the output DLL are correct, you may verify them using [Dependency Walker](#). When you load the DLL, the correct function names should appear in Dependency Walker as shown below.

| E | Ordinal ^ | Hint | Function | Entry Point |
|---|------------|------------|---------------------------|-------------|
|  | 1 (0x0001) | 0 (0x0000) | XSplitScriptPluginCall | 0x0008D4A4 |
|  | 2 (0x0002) | 1 (0x0001) | XSplitScriptPluginDestroy | 0x0008E633 |
|  | 3 (0x0003) | 2 (0x0002) | XSplitScriptPluginInit | 0x00090A64 |

If the names were **not** exported correctly, they may appear like either of the following:

| E | Ordinal ^ | Hint | Function | Entry Point |
|---|------------|------------|------------------------------|-------------|
|  | 1 (0x0001) | 0 (0x0000) | _XSplitScriptPluginCall@20 | 0x00003C30 |
|  | 2 (0x0002) | 1 (0x0001) | _XSplitScriptPluginDestroy@0 | 0x00003AF0 |
|  | 3 (0x0003) | 2 (0x0002) | _XSplitScriptPluginInit@0 | 0x00003AE0 |

| E | Ordinal ^ | Hint | Function | Entry Point |
|---|------------|------------|--|-------------|
|  | 1 (0x0001) | 0 (0x0000) | ?XSplitScriptPluginCall@@YGHPAUIXSplitScriptDllContext@@PA_WPAPA_WI2@Z | 0x00003CE0 |
|  | 2 (0x0002) | 1 (0x0001) | ?XSplitScriptPluginDestroy@@YGXXZ | 0x00003B60 |
|  | 3 (0x0003) | 2 (0x0002) | ?XSplitScriptPluginInit@@YGHXZ | 0x00003B50 |

Testing a custom DLL in XSplit Broadcaster

1. Make sure developer mode is enabled in XSplit Broadcaster. This will allow XSplit to load your DLL even if it is unsigned, which allows you to test and debug your DLL during development. You can find this in the Tools menu > General Settings... > Advanced tab > Enable developer mode. Close XSplit before proceeding to the next step.
2. If your DLL is dependent on other DLLs, place those dependencies in the XSplit Broadcaster installation folder.
3. Place your custom DLL in the **Scriptdlls** subdirectory.
4. Run XSplit Broadcaster. Your DLL will have been loaded and XSplitScriptPluginInit() will have been called by the that time the application's splash screen has closed.
5. To verify that your DLL has been loaded, you may use a tool such as [Process Explorer](#) or [ListDLLs](#).