GO-GLOBAL

Component SDK

Component Software Development Kit for Windows and Linux

Version 4.8.2

COPYRIGHT AND TRADEMARK NOTICE

Printed in the United States of America.

CONTENTS

1 INTRODUCTION TO THE COMPONENT SDK

About this SDK

This SDK includes everything you need to develop your own GO-Global Component.

This folder…	Contains…
Component_SDK\sample	Samples of how the SDK can be used
Component_SDK\sample\include	The header files needed to build a Component csx.h
Component_SDK\sample\lib	The following libraries: cs4.lib, cs4s3.lib, utl.lib
Component_SDK\doc	The Component SDK document

What Can You Do With the SDK?

The information in this document and the header files in the GO-Global Component Software Development Kit can be used to create a custom Component for your GO-Global application.

What is a GO-Global Component?

A GO-Global Component is a pair of Client/Server modules that communicate across a channel. The Server module is a Windows DLL that can communicate with the Client module. The Client module can be a Windows DLL or a Linux SO library. GO-Global provides a framework for loading these dynamic modules on the various platforms.

GO-Global is an application Server that delivers the functionality of Windows applications to virtually any desktop computer, including Macintosh, UNIX, Linux, or Windows. When running GO-Global, a user can interact with applications running on a remote Server computer. Mouse and key events are transmitted to the Server, where they are processed. Display updates resulting from the user events are transmitted from the Server back to the desktop terminal. With GO-Global, applications are deployed, managed, and executed on the Server. The Client is a very simple application that displays application user interfaces and forwards mouse and key events to the Server.

During program execution, the Component Server manages all Client/Server communication between GO-Global processes. The Component Server frames application requests, optionally compresses and encrypts the requests, and handles low-level network IO operations.

GO-Global includes such pre-made Components as: **fileio**, that lets you save a file to a drive on the Client computer; **clipboard**, that lets you copy a selection to the Client clipboard for use in an application running on the Client computer; and **printing**, that lets you print a document using a printer attached to the Client computer.

The Component SDK provides a way to extend the capabilities of an application running on a GO-Global server beyond the Components already included with GO-Global. With a custom GO-Global Component, an application running on the GO-Global Server can communicate directly with the GO-Global client, and vice-versa. For instance, the application could request information from the client or display a message to the user through the communication channels that the SDK provides.

2 CREATING THE SERVER-SIDE DLL

In order to create a GO-Global component, you must create two plug-ins, one for the Client-side, and one for the Server-side.

This section contains information that will help you create the Server-side DLLs for your component.

Creating the Server-Side DLL

The Server-side plug-in DLL has to link with the following libraries:

- cs4s3.lib
- utl.lib

Registering/Unregistering the Components

In your DLLs, register components using **CS_RegisterClass**. Unregister components using **CS_UnregisterClass**.

CS_RegisterClass
In the DllMain function, call this function from the **DLL_PROCESS_ATTACH** case.

In this function, pass the pointers to the three callbacks (**CreateImplCB**, **DeleteImplCB**, and **ProcessRequestCB**), a name by which the system recognizes the component, and the component's **CS_VERSION**.

Naming the Component

In the Server DLL, the name will be the same as the first name passed to **CS_InitComponent**. For more information, see **CS_InitComponent** in section 4: The C Functions.

In the Client DLL, the name will be the same as the second name passed to **CS_InitComponent**. For more information, see **CS_InitComponent** in section 4: The C Functions.

CS_InitComponent is only called in the Server DLL.

Function Prototype

```
CS_DllExport void CS_RegisterClass(CS_Char *pName, CreateImplCB, DeleteImplCB,
      ProcessRequestCB, CS_VERSION* version );
```

Parameters

`CS_Char *pName`	pointer to pName
`CreateImplCB`	passes pointer to CreateImpl
`DeleteImplCB`	passes pointer to DeleteImpl
`ProcessRequestCB`	passes pointer to ProcessRequest
`CS_VERSION* version`	version information for the component

CS_UnregisterClass

In the DllMain function, call this function from the **DLL_PROCESS_DETACH** case.

Function Prototype

```
CS_DllExport      void  CS_UnregisterClass(CS_Char *pName);
```

Parameters

`CS_Char *pName`	pointer to pName

Sample DllMain

You will have to create a DllMain section in your Server-side plug-in. Here is a sample DllMain that does everything typically required:

```
/**
 * File: DllMain.cpp
 *
 */


#include <windows.h>
#include <stdio.h>
#include "MyTestServer.h"
```

```
MyServer* g_pMyServer = NULL;

void processRequestCB (void* pServerObj, int requestId, CSChannelPtr channel)
{
  MyServer* pServer = (MyServer*) pServerObj;
  pServer->processRequest (requestId, channel);
}

void* createImplementation (CSComponentPtr pComp)
{
  MyServer* pServer = new MyServer;

  if (pServer)
  {
    pServer->setComponent (pComp);
  }

  return (pServer);
}

void deleteImplementation (void *pImpl)
{
  delete ((MyServer*) pImpl);
}

#ifdef WIN32
BOOL WINAPI DllMain (HANDLE hInstance, DWORD reason, void* lpReserved)
{
```

```
switch (reason)
{
    case DLL_PROCESS_ATTACH:
    {
      CS_Initialize();

      CS_VERSION version;
      version.major    = 4;
      version.minor    = 1;
      version.revision  = 3;
      version.build    = 0;

      CS_RegisterClass (
          L"GraphOn::RXP::MyTestServerProxy",
          createImplementation,
          deleteImplementation,
          processRequestCB
          &version);

      break;
    }

    case DLL_PROCESS_DETACH:
    {
      CS_UnregisterClass (L"GraphOn::RXP::MyTestServerProxy");

      if (g_pMyServer)
      {
```

```
            delete g_pMyServer;
            g_pMyServer = NULL;
        }

        break;
      }

      case DLL_THREAD_ATTACH:
        break;

      case DLL_THREAD_DETACH:
        break;
  }

  return TRUE;
}
#endif
```

processRequest Callback and CS_PrepareRequest

To communicate between the Client and the Server, create a **processRequestCB** function.

A pointer to the implementation object that you create with the **createImplementation**() callback function is passed to **processRequestCB**. This lets you pass the request on to your implementation class: Pass the **processRequestCB** function to **CS_RegisterClass**(). GO-Global calls this function when it has a request to process. On the Server side, this request originates from the Client. On the Client side, this request originates from the Server. You specify the request yourself using **CS_PrepareRequest**. Use **CS_PrepareRequest** to prepare a request of a certain ID known to the module you are requesting from. Then, in the **processRequest** method of your implementation class, create a case for that ID.

Synchronous/Asynchronous Request

Synchronous requests between the Client and Server are accomplished by using the **CS_WaitForReply** function after sending the request. Using **CS_WaitForReply** makes sure that the reply comes back before continuing execution of the program. If you don't use the **CS_WaitForReply** function, the result will be asynchronous communication. Assuming the request went through, the code will keep executing.

Server-Side Exports

While creating your Server side DLL, you have to export certain functions. On the Server side, you must export the following functions:

- GetPluginType
- InitComponent
- CleanupComponent

Export a GetPluginType Function

Export a **GetPluginType** function that returns **PLUGIN_COMPONENT**.

This value is defined in csx.h.

```
INT WINAPI GetPluginType()
{
  return (PLUGIN_COMPONENT);
}
```

- Make sure it is exported from your DLL. For example, you could add it to the project's .def file.

Export the InitComponent Function

```
MyServer *g_pMyServer = NULL;

VOID WINAPI InitComponent()
```

```
{
  CSComponentPtr pComp = CS_InitComponent (
      L"GraphOn::RXP::MyTestServer",
      L"GraphOn::RXP::MyTestClient");

  if (pComp)
  {
      g_pMyServer = (MyServer*) CS_GetLocalServer (pComp);
  }
}
```

- Make sure it is exported from your DLL. For example, you could add it to the project's .def file.

Export the CleanupComponent Function

```
VOID WINAPI CleanupComponent()
{
  // Delete and clean up.
  // DO NOT delete g_pMyServer. A callback will do that for you.
}
```

- Make sure it is exported from your DLL. For example, you could add it to the project's .def file.

Note: *For information about how to set up the DLL to be used by GO-Global, refer to section 5: Setting up the Plug-ins.*

3 CREATING THE CLIENT-SIDE PLUG-IN

The Client module can be a Win32 DLL or a Linux SO library. This section contains information about creating the plug-in on the Client-side.

Windows

Creating the Client-Side DLL

To make your Client component work on Microsoft Windows XP or higher, develop it using only Win32 API functions available in Microsoft Windows XP.

The Client-side plug-in DLL has to link with the following library:

1. cs4.lib
2. utl.lib

For information about other functions you can use in your Windows plug-in, see section 4: The C Functions.

Sample DllMain

You will have to create a DllMain section in your Client-side plug-in. Here is a sample DllMain that does everything typically required:

```
/**
 * File: DllMain.cpp
 *
 */

#include <windows.h>
#include <stdio.h>
#include "MyTestClient.h"

void processRequestCB (void *pClientObj, int requestId, CSChannelPtr channel)
{
```

```
  MyTestClient* pClient = (MyTestClient*) pClientObj;
  pClient->processRequest (requestId, channel);
}

void* createImplementation (CSComponentPtr pComp)
{
  MyTestClient  *pClient = new MyTestClient;

  if (pClient)
  {
    pClient->setComponent (pComp);
  }

  return (pClient);
}

void deleteImplementation (void* pImpl)
{
  delete ((MyTestClient *)pImpl);
}

#ifdef WIN32
BOOL WINAPI DllMain(HANDLE hInstance, DWORD reason, void* lpReserved)
{
  switch (reason)
  {
      case DLL_PROCESS_ATTACH:
      {
```

```
        CS_Initialize ();

        CS_RegisterClass (
            L"GraphOn::RXP::MyTestClient",
            createImplementation,
            deleteImplementation,
            processRequestCB
            &version);
        break;
      }

      case DLL_PROCESS_DETACH:
      {
        CS_UnregisterClass (L"GraphOn::RXP::MyTestClient");
        CS_Shutdown();
        break;
      }

      case DLL_THREAD_ATTACH:
        break;

      case DLL_THREAD_DETACH:
        break;
  }

  return TRUE;
}
#endif
```

Client-Side Exports

On the Client side, there is one function to export.

Export a GetPluginType Function

Export a **GetPluginType** function that returns **PLUGIN_COMPONENT**.

This value is defined in csx.h.

```
INT WINAPI GetPluginType()
{
  return(PLUGIN_COMPONENT);
}
```

- Make sure it is exported from your DLL.

***Note**: For information about how to set up the DLL to be used by GO-Global, refer to section 5: Setting up the Plug-ins.*

Linux

To create the Client-side plug-in to run on a Linux computer, follow the instructions for creating the plug-in for Windows, except for the following: Take all the code that was in the **DLL_PROCESS_ATTACH** case in DllMain, and put it into its own function named dlStart. Take all the code that was in the **DLL_PROCESS_DETACH** case in DllMain, and put it into its own function named dlFinish. Make sure the functions dlStart, dlFinish, and **GetPluginType**() are all visible outside the plugin by using extern "C".

dlStart Sample Code

```
extern "C" void dlStart()
{
  CS_Initialize();

  CS_VERSION version;
  version.major    = 4;
  version.minor    = 1;
  version.revision  = 3;
  version.build    = 0;

  CS_RegisterClass(
      L"GraphOn::RXP::MyTestClient",
      createImplementation,
      deleteImplementation,
      processRequestCB,
      &version);
}
```

dlFinish Sample Code

```
extern "C" void dlFinish()
{
  CS_UnregisterClass (L"GraphOn::RXP::MyTestClient");
  CS_Shutdown();
}
```

4 THE C FUNCTIONS

CS_Initialize

In the DllMain function, call this function from the **DLL_PROCESS_ATTACH** case (Win32) or from the dlStart function (Linux).

Function Prototype

```
CS_DllExport      void  CS_Initialize(void);
```

CS_Shutdown

This function is to be used in the Client DLL only.

In the DllMain function, call this function from the **DLL_PROCESS_ATTACH** case (Win 32) or from the dlFinish function (Linux).

Function Prototype

```
CS_DllExport      void  CS_Shutdown(void);
```

CS_RegisterClass

In the DllMain function, call this function from the **DLL_PROCESS_ATTACH** case (Win32) or from the dlStart function (Linux).

In this function, pass the pointers to the three callbacks (**CreateImplCB**, **DeleteImplCB**, and **ProcessRequestCB**), a name by which the system recognizes the component, and the component's **CS_VERSION**.

This is used in both the Server DLL and Client (Win32 DLL and Linux so).

Naming the Component

In the Server DLL, the name will be the same as the first name passed to **CS_InitComponent**. For more information, see **CS_InitComponent** below.

In the Client DLL, the name will be the same as the second name passed to **CS_InitComponent**. For more information, see **CS_InitComponent** below.

CS_InitComponent is only called in the Server DLL.

Function Prototype

```
CS_DllExport void CS_RegisterClass (CS_Char* pName, CreateImplCB,
      DeleteImplCB, ProcessRequestCB, CS_VERSION* version);
```

Parameters

`CS_Char *pName`	pointer to pName
`CreateImplCB`	passes pointer to CreateImpl
`DeleteImplCB`	passes pointer to DeleteImpl
`ProcessRequestCB`	passes pointer to ProcessRequest
`CS_VERSION* version`	version information for the component

CS_UnregisterClass

In DllMain, call this function from the **DLL_PROCESS_DETACH** case (Win32) or from the dlFinish function (Linux). Use the same name that you used in **CS_RegisterClass**().

Function Prototype

```
CS_DllExport void CS_UnregisterClass (CS_Char* pName);
```

Parameters

`CS_Char *pName`	pointer to pName

CS_InitComponent

CS_InitComponent is only called in the Server DLL.

Call the **CS_InitComponent** function from the **InitComponent** function. GO-Global will search for the **InitComponent** function in the DLL.

The **InitComponent** function must be in the .def file of your DLL, in the format below.

CS_InitComponent returns a pointer that must be passed to **CSComponentPtr**. Use this pointer to call **CS_GetLocalServer**, which will then return the implementation object created in the **CreateImplCB** callback. Save the implementation object pointer in a global variable for later use. You may also save the **CSComponentPtr** into your implementation object.

Function Prototype

```
CS_DllExport void* CS_InitComponent (CS_Char* pLocal, CS_Char* pRemote);
```

Parameters

`CS_Char *pLocal`	pointer to pLocal
`CS_Char *pRemote`	pointer to pRemote

CS_SetLastError

The **CS_SetLastError** function tells GO-Global about the most recent error that occurred.

Function Prototype

```
CS_DllExport void CS_SetLastError (int iError);
```

Parameters

```
int iError
```

CS_GetChannel

CS_GetChannel retrieves the component's channel, which is needed in several functions, for example, **CS_PrepareRequest**.

This information is also used to transfer data between the Client and Server DLLs. See the read/write channel functions below:

Read Channel Functions	Write Channel Functions
CS_CH_ReadFully	CS_CH_WriteFully
CS_CH_ReadByte	CS_CH_WriteFully2
CS_CH_ReadBoolean	CS_CH_WriteByte
CS_CH_ReadShort	CS_CH_WriteBoolean
CS_CH_ReadUnsignedShort	CS_CH_WriteShort
CS_CH_ReadInt	CS_CH_WriteUnsignedShort
CS_CH_ReadUTF	CS_CH_WriteChar
CS_CH_ReleaseUTF	CS_CH_WriteInt
	CS_CH_WriteUnsignedInt
	CS_CH_WriteLong
	CS_CH_WriteUTF

When finished with the channel, call **CS_ReleaseChannel** to release the channel.

Function Prototype

```
CS_DllExport CSChannelPtr CS_GetChannel(CSComponentPtr);
```

Parameters

`CSComponentPtr`	points to your CSComponent

CS_ReleaseChannel

CS_ReleaseChannel is used to release a channel retrieved with the **CS_GetChannel** function.

Function Prototype

```
CS_DllExport void CS_ReleaseChannel(CSComponentPtr,     CSChannelPtr);
```

Parameters

`CSComponentPtr`	points to your CSComponent
`CSChannelPtr`	points to your CSChannel

CS_GetLocalServer

CS_GetLocalServer is used to query a component for its implementation object.

Function Prototype

```
CS_DllExport      void  *CS_GetLocalServer(CSComponentPtr);
```

Parameters

`CSComponentPtr`	points to your CSComponent

CS_PrepareRequest

CS_PrepareRequest must be called before writing to a channel.

First, specify the request ID in the **IRequest** parameter. Then call the appropriate write channel functions to pass the data. Last, for synchronous communication, call **CS_WaitForReply** to wait for the reply, then use the channel read functions to read the reply (if any).

Function Prototype

```
CS_DllExport void CS_PrepareRequest(CSComponentPtr, CSChannelPtr, long
      lRequest);
```

Parameters

`CSComponentPtr`	points to your CSComponent
`CSChannelPtr`	points to your CSChannel
`long lRequest`	refers to the request identifier

CS_WaitForReply

CS_WaitForReply is called once all the data has been written to a channel. Call **CS_WaitForReply** then use the read channel functions to read the reply. Used for synchronous communication.

When all the data is read, call **CS_ReleaseChannel** to release the channel**.**

Function Prototype

```
CS_DllExport void CS_WaitForReply (CSComponentPtr, CSChannelPtr);
```

Parameters

`CSComponentPtr`	points to your CSComponent
`CSChannelPtr`	points to your CSChannel

CS_PrepareReply

CS_PrepareReply is usually only used in the Client DLL.

CS_PrepareReply is called to prepare a reply. Then the write channel functions are called to write out the reply. After the reply is written out, **CS_CH_Flush** is called to flush the channel.

Function Prototype

```
CS_DllExport void CS_PrepareReply(CSComponentPtr, CSChannelPtr);
```

Parameters

`CSComponent`	points to your CSComponent
`CSChannelPtr`	points to your CSChannel

CS_CH_Flush

Use **CS_CH_Flush** to flush a channel after receiving data and calling a **CS_PrepareReply**. It may also be used after a **CS_PrepareRequest** before releasing the channel.

Function Prototype

```
CS_DllExport bool CS_CH_Flush (CSComponentPtr, CSChannelPtr,
        bool fRelease);
```

Parameters

`CSComponentPtr`	points to your CSComponent
`CSChannelPtr`	points to your CSChannel
`bool fRelease`	indicates if the channel will be released

CS_CH_ReadFully

CS_CH_ReadFully reads exactly lLen bytes from the data input stream.

Function Prototype

```
CS_DllExport void CS_CH_ReadFully(CSChannelPtr ptr,
        unsigned char* puc, long lOff, long lLen);
```

Parameters

`CSChannelPtr ptr`	points to your CSChannel
`unsigned char *puc`	points to the buffer into which the data is read
`long lOff`	indicates where to start offset of the data
`long lLen`	indicates the number of bytes to read

CS_CH_ReadByte

CS_CH_ReadByte reads exactly one byte from the data input stream.

Function Prototype

```
CS_DllExport char CS_CH_ReadByte(CSChannelPtr);
```

Parameters

```
CSChannelPtr      points to your CSChannel
```

CS_CH_ReadBoolean

CS_CH_ReadBoolean reads exactly one boolean from the data input stream.

Function Prototype

```
CS_DllExport bool CS_CH_ReadBoolean (CSChannelPtr);
```

Parameters

```
CSChannelPtr      points to your CSChannel
```

CS_CH_ReadShort

CS_CH_ReadShort reads exactly one short from the data input stream.

Function Prototype

```
CS_DllExport short CS_CH_ReadShort (CSChannelPtr);
```

Parameters

```
CSChannelPtr      points to your CSChannel
```

CS_CH_ReadUnsignedShort

CS_CH_ReadUnsignedShort reads exactly one unsigned short from the data input stream.

Function Prototype

```
CS_DllExport unsigned short CS_CH_ReadUnsignedShort    (CSChannelPtr);
```

Parameters

`CSChannelPtr`	points to your CSChannel

CS_CH_ReadInt

CS_CH_ReadInt reads exactly one int from the data input stream.

Function Prototype

```
CS_DllExport int CS_CH_ReadInt (CSChannelPtr);
```

Parameters

`CSChannelPtr`	points to your CSChannel

CS_CH_ReadLong

CS_CH_ReadLong reads exactly one __int64 from the data input stream.

Function Prototype

```
CS_DllExport __int64 CS_CH_ReadLong(CSChannelPtr);
```

Parameters

`CSChannelPtr`	points to your CSChannel

CS_CH_ReadUTF

CS_CH_ReadUTF reads one UTF string from the data input stream. After **CS_CH_ReadUTF**, use **CS_CH_ReleaseUTF** to release the allocated memory.

Function Prototype

```
CS_DllExport unsigned short* CS_CH_ReadUTF (CSChannelPtr);
```

Parameters

`CSChannelPtr`	points to your CSChannel

CS_CH_ReleaseUTF

CS_CH_ReleaseUTF releases the UTF string memory allocated by **CS_CH_ReadUTF**.

Function Prototype

```
CS_DllExport void CS_CH_ReleaseUTF (CSChannelPtr,
  Unsigned short*);
```

Parameters

`CSChannel`	points to your CSChannel
`unsigned short *`	points to the UTF to be released

CS_CH_WriteFully

CS_CH_WriteFully writes an array of lCount bytes to the data stream. Use **CS_CH_WriteFully**, not **CS_CH_WriteFully2**, when you want to start writing the data from the beginning.

Function Prototype

```
CS_DllExport void CS_CH_WriteFully (CSChannelPtr ptr,
  const unsigned char* puc, long lCount);
```

Parameters

`CSChannelPtr ptr`	points to your CSChannel
`const unsigned char* puc`	points to the buffer of data to write to
`long lCount`	indicates the number of bytes to write

CS_CH_WriteFully2

CS_CH_WriteFully2 writes an array of lCount bytes to the data stream. Using **CS_CH_WriteFully2**, as opposed to **CS_CH_WriteFully**, lets you start writing the data somewhere other than the beginning.

Function Prototype

```
CS_DllExport void CS_CH_WriteFully2 (CSChannelPtr ptr,
  const unsigned char* puc, long lOffset, long lCount);
```

Parameters

`CSChannelPtr ptr`	points to your CSChannel
`const unsigned char* puc`	points to the buffer of data to write to
`long lOffset`	indicates the start of the offset of the data
`long lCount`	indicates the number of bytes to write

CS_CH_WriteByte

CS_CH_WriteByte writes exactly one byte to the data stream.

Function Prototype

```
CS_DllExport void CS_CH_WriteByte(CSChannelPtr ptr, char);
```

Parameters

`CSChannelPtr ptr`	points to your CSChannel
`char`	indicates a character

CS_CH_WriteBoolean

CS_CH_WriteBoolean writes exactly one boolean to the data stream.

Function Prototype

```
CS_DllExport void CS_CH_WriteBoolean(CSChannelPtr, bool);
```

Parameters

`CSChannelPtr ptr`	points to your CSChannel
`bool`	indicates a boolean

CS_CH_WriteShort

CS_CH_WriteShort writes exactly one short to the data stream.

Function Prototype

```
CS_DllExport      void  CS_CH_WriteShort (CSChannelPtr , short);
```

Parameters

`CSChannelPtr`	points to your CSChannel
`short`	indicates a short

CS_CH_WriteUnsignedShort

CS_CH_WriteUnsignedShort writes exactly one unsigned short to the data stream.

Function Prototype

```
CS_DllExport void CS_CH_WriteUnsignedShort (CSChannelPtr, unsigned
      short);
```

Parameters

`CSChannelPtr`	points to your CSChannel
`unsigned short`	indicates an unsigned short

CS_CH_WriteChar

CS_CH_WriteChar writes exactly one character to the data stream.

Function Prototype

```
CS_DllExport void CS_CH_WriteChar(CSChannelPtr, char);
```

Parameters

`CSChannelPtr`	points to your CSChannel
`char`	indicates a character

CS_CH_WriteInt

CS_CH_WriteInt writes exactly one integer to the data stream.

Function Prototype

```
CS_DllExport void CS_CH_WriteInt (CSChannelPtr, int);
```

Parameters

`CSChannelPtr`	points to your CSChannel
`int`	indicates an integer

CS_CH_WriteUnsignedInt

CS_CH_WriteUnsignedInt writes exactly one unsigned integer to the data stream.

Function Prototype

```
CS_DllExport void CS_CH_WriteUnsignedInt (CSChannelPtr,
      unsigned int);
```

Parameters

`CSChannelPtr`	points to your CSChannel
`unsigned int`	indicates an unsigned integer

CS_CH_WriteLong

CS_CH_WriteLong writes exactly one long value to the data stream.

Function Prototype

```
CS_DllExport void CS_CH_WriteLong (CSChannelPtr, long);
```

Parameters

`CSChannelPtr`	points to your CSChannel
`long`	indicates a long value

CS_CH_WriteUTF

CS_CH_WriteUTF writes exactly one byte to the data stream.

Function Prototype

CS_DllExport void **CS_CH_WriteUTF**(CSChannelPtr, const unsigned short *);

Parameters

`CSChannelPtr`	points to your CSChannel
`const unsigned short`	points to the UTF to be written

5 SETTING UP THE PLUG-INS

When you are finished creating your Server-side and Client-side Component Server plug-ins, you must set them up on the computers running GO-Global.

No matter what operating system and/or plug-in, the method to set up your Server-side plug-in is the same. The method you use to set up your Client-side plug-in will be determined by the operating system you use, and if you are running GO-Global as a plug-in with Mozilla Firefox or Microsoft ActiveX.

Working on a single computer

You can use your computer as both a Client and Server during development. Under the Registry key HKEY_LOCAL_MACHINE\SOFTWARE\GraphOn\GO-Global\AppServer, there are two keys that hold the location of the Client and Server DLLs. If the keys are not there, see the sections below for the directories for the Client and Server DLLs. The keys are:

- ClientPluginsPath
- ServerPluginsPath

Note*: On a 64-bit client, relocate the* ***ClientPluginsPath*** *registry entry to the following key on the client: HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\GraphOn\GO-Global*

Setting up a Plug-in on the Server

1. Place your completed Server-side plug-in DLL in the "/Server/plugins/" folder, where "Server" represents the folder where your Server application is stored.
2. Make sure that your application links to the plug-in so that the plug-in will load at application startup.

Setting up a Plug-in on the Client Computer – Microsoft Windows

If you are *not* running GO-Global as a Plug-in, place your completed Client-side plug-in DLL in the "/gg/plugins/" folder, where "gg" represents the folder where gg-client.exe is stored.

If you are running Global as a Microsoft ActiveX Plug-in, place your completed Client-side plug-in DLL in the "/Windows/Downloaded Program Files/plugins/" folder, where "Windows" represents the folder where Microsoft Windows is stored.

Note: *If there is no plugins folder under the "/Windows/Downloaded Program Files/" folder, create one.*

If you are running GO-Global as a Mozilla Firefox Plug-in, place your completed Client-side plug-in DLL in "Mozilla/Programs/Plugins/Plugins/" where "Mozilla" represents the folder where the Mozilla Firefox is stored.

Note: *The path of the folder where Mozilla Firefox is stored will depend on your version of Firefox.*

Setting up a Plug-in on the Client Computer – Linux

If you are not running GO-Global as a Plug-in, place your completed Client-side plug-in DLL in the "/gg/plugins/" folder, where "gg" represents the folder where the GO-Global application is stored.

If you are running GO-Global as a Mozilla Firefox Plug-in, place your completed Client-side plug-in DLL in "Mozilla/Programs/Plugins/Plugins/" where "Mozilla" represents the folder where the Mozilla Firefox is stored.

Note: *The path of the folder where Mozilla Firefox is stored will depend on your version of Firefox*.