# 3dfx Interactive

## Central Services API Reference Document

# 0 Contents

## 0.1 Revision History

| Revision | Description | Date |
|---|---|---|
| 0.1 | --Draft received from Chris Dow | 000209 |
| 0.2 | --Reformatted to make maintenance easier. Miscellaneous omissions and errors corrected. “Issues” section added. | 000209 |
| 0.3 | --Fixed omission in csGetCSContext, added CS_BUFFER_ZBUFFER and a few new error codes to csAlloc, Made 3DLFB in csLock a flag, just like …READ and …WRITE, and fixed miscellaneous errata. Also changed the naming schemes of some of the macros, to make them more consistent. Didn’t un-highlight the changes from 0.2, since 0.2 never made it into StarTeam. | 000214 |
| 0.4 | (No description of changes) | 000224 |
| 0.5 | Major overhaul of document to reflect evolutionary changes in the spec. | 000609 |
| | | |
| | | |
| | | |

## 0.2 Issues

- Need to add an explanation of the encapsulating transport structure
- When a lock  is held on a surface across a mode-change whatever area of the LFB the writing was going to may cause minor display corruption.

# 1 Introduction

Our business requires fully functional GDI, DirectDraw™, Direct3D™, and OpenGL™ drivers on Windows 98 and NT; fully functional X and OpenGL drivers on Linux; and fully functional QuickDraw™ and OpenGL drivers on the Macintosh. This means that all must work concurrently, and changes one piece makes to the system must be handled gracefully by the other pieces.

The intended audience of this document is software engineers who need to understand the Central Services API.

## 1.1 Historical Context

In the past, this was done by cooperation between the Windows driver software and Glide/OpenGL. This worked reasonably well (with some problems having to do with tiled vs. linear memory) for the case where Glide is cooperating with the Windows driver software by writing a command buffer then making a call through ExtEscape() to execute it. However, when Glide operated in full-screen mode—needing full control of the hardware—we encountered unsolvable problems.

These problems arose from Glide's need to have exclusive access to the hardware for some period of time. During this time, Glide reprogrammed the command FIFO hardware to monitor writes to an area of memory. There was no coordination with the display driver, and if a catastrophic event like an alt-tab or a BSOD occurred there was no mutual understanding of memory layout in the frame buffer. This sometimes resulted in Glide depositing command traffic on the desktop, and sometimes even system hangs. The workaround for this problem is unacceptable to many parties—Microsoft's WHQL team being one of them.

Thus, we concluded that there needs to be a more cooperative way for APIs that do not reside in the display driver to access the hardware.

## 1.2 Central Services Clients

Any API that enables hardware-accelerated rendering (GDI, Direct3D, OpenGL, Glide, X, and QuickDraw) or that enables applications to write to video memory (DirectDraw) must use *some* piece of Central Services. However, the level of utilization will differ between APIs in the display driver and APIs that exist outside it. For example, the Central Services driver interface will use the same pieces of code in the display driver that other subsystems may already use, while all subsystems will use the memory allocator.

### 1.2.1 Client state machine

The following is a rough outline of the general flow and operation of a Central Services client.

- Call csInit() to initialize the Central Services interface, and establish a connection to the server.
- Call csGetProtocolRevision() to determine compatibility level with the server.
- Call csGetGraphicalContext() to associate a CS context with the client's window, and to register a state callback function that Central Services will use to inform the client of asynchronous events that affect the operation of the client.
- Call csAlloc() to allocate any and all persistent buffers that will be needed during the client's lifetime. Persistent buffers include command fifo buffers, and sentinel buffers. (Note: although it is not absolutely necessary to claim all these buffers at the outset, it is highly advisable that clients do so, prior to allocating other types of buffers. Otherwise, the client may not be able to obtain all of the persistent space it requires.)
- Call csAcquireOverlay(), to attempt to acquire an available hardware overlay resource. While not completely necessary for operation, using an overlay resource can greatly increase the rasterization performance of what the client renders.

  Assuming that the underlying hardware has overlay capabilities, the only way a CS client will fail to obtain this overlay resource is if another software entity (CS client or otherwise) already has it. However, that other entity will lose its overlay if the first CS client: (rvb)

  - ***Obtains the window focus***
  - ***Achieves exclusivity***

  But since one of the requirements of Exclusive Mode is ownership of focus, this essentially means that all that is required for ownership of the overlay is for the window to have focus. Unfortunately, due to the asynchronous nature of the focus message, the client can't track this state itself; Central Services must do this on its behalf. (rvb)

  ***Double buffering and overlays:*** Perhaps the only downside to overlays is that they complicate double buffering a little, and the clients must be smart about reading the rendered surface, when an overlay resource is used. The following is a brief description of how double buffering must be done in various circumstances:

- ***Exclusive Mode, no overlay:*** The client's window region on the primary is considered the front buffer. The content of the off-screen back buffer is blitted to the front buffer, against a clip-list, and preferably during vertical retrace.

  One might think one could simply change the hardware's "start of primary" offset register each frame, to alternate between two identically-sized buffers. Ordinarily this would be a keen idea, as it would allow us to avoid blitting, even when the underlying hardware doesn't have an overlay resource. However, two key factors make this a Bad Idea™:

  - If the client were to lose exclusivity, it would have to guarantee that it exits Exclusive Mode with the "start of primary" offset register pointing at the primary surface. Otherwise the other apps would end up drawing to the wrong area of the frame buffer.
  - Even if the client could make the preceding guarantee, it's still a Bad Idea™ on MS Windows platforms. A few OpenGL apps utilize GDI for their menu subsystems, and GDI was not designed with double buffering in mind. Thus, half of what GDI attempted to draw would end up on the back buffer, and half would end up on the front buffer. Not good.
    - ***Exclusive Mode, with overlay:*** For all intents and purposes, this can only be done via triple buffering. The front buffer is the destination rectangle of the overlay resource, which (hopefully!) coincides with the entire primary surface. The "start of overlay source" register is changed each frame (preferably at vsync time) to alternate between the two identically-sized back buffers. The primary surface must be filled with the correct colorkey value. The WM_PAINT message handler must fill in the specified regions with the colorkey value, as well.
    - ***Windowed Mode, no overlay:*** The client's window region on the primary is considered the front buffer. The content of the off-screen back buffer is blitted to the front buffer, against a clip-list, and preferably during vertical retrace.
    - ***Windowed Mode, with overlay:*** For all intents and purposes, this can only be done via triple buffering. The front buffer is the destination rectangle of the overlay resource, which (hopefully!) coincides with the client region of the CS client's target rendering window. The "start of overlay source" register is changed each frame (preferably at vsync time) to alternate between the two identically-sized back buffers.

- Call csAlloc() to acquire any back buffers, z-buffers, texture heaps, etc., that are known to be needed for the rendering process.

- Render a frame, or a portion thereof. The output of the CS client's rendering process is command fifo traffic, which the client writes sequentially into a (previously allocated) command fifo buffer. When the command fifo buffer is almost full, the client should "top it off" with a command packet that writes a serial number into a (previously allocated) sentinel buffer.

- Call csExecuteCommands(), to request that the server insert the command packets into the underlying hardware's command fifo. (The server may accomplish this by either a direct memcpy, or by appending a "RET" command packet to the end of the fifo buffer, and inserting a "JSR" packet into the actual command fifo, instructing the hardware to treat the command buffer as a subroutine.)

  Because multiple processes could be sharing the same hardware resources, it's possible for the hardware state to become dirty.

## 1.3 Litmus Tests

*Central Services*' charter is to be the referee between its clients and other discrete, independent subsystems, all competing for the same resources. But it aims to fulfill this responsibility with a minimal amount of interference; it tries to stay out of its clients' way, whenever possible, only getting involved if it is necessary to do so.

With this charter in mind, the following litmus tests for new functionality fomented in the minds of its creators, during the evolution of this specification. These litmus tests are what guide the decision to add (or refrain from adding) a new responsibility to *Central Services*.

### 1.3.1 Hardware Litmus Test

- Does the task involve the use of a limited or shared resource that requires arbitration for fair use between competing clients?
- Is there ***no*** safe mechanism or methodology by which the clients can perform this arbitration process amongst themselves and on their own, without danger of race conditions or other forms of resource contention? (rvb)

If the answer to ***both*** questions is ***yes***, then the task ***fails*** the hardware litmus test, and therefore falls under the auspices of the *Central Services* charter. Thus, it should be handled by the *Central Services* server on behalf of its clients.

Now, this is not to say that Central Services needs to (or even should) handle all of it's responsibilities the same way. For any given resource that requires arbitration, Central Services could choose to use any of the following, (or others) based upon performance needs:

- OS-specific call gate (i.e. ExtEscape on Win32, Server extension on X)
- Named pipe, provided by Central Services on behalf of the clients
- Shared memory region, provided by Central Services on behalf of the clients

### 1.3.2 Software Litmus Test

This is really more of a guideline than a litmus test. If the source code for a client is intended to be usable under multiple operating systems, but one of those operating systems has a particular quirk about it that has to be special cased, it may make sense to hide that quirk under the abstractions provided by the Central Services API. (i.e., let CS handle it.) However, it doesn't seem prudent to hide "one-offs" at the expense of performance. So these cases will have to be dealt with individually.

## 1.4 Requirements

The purpose of this section is to simply state the needs that Central Services must fulfill. Once these are clear, this section can be used to guide later steps in the design process.

All APIs that use hardware to accelerate 3D graphics perform the following services:

- Get info about the system configuration
- Get some memory for rendering and textures
- Initialize hardware for rendering
- Initialize command transport to enable rendering
- Initialize video for rendering
- Get rendered pixels to the screen
- Generate and execute rendering commands
- Occasionally read from the hardware to make sure it's performed requested operations
- Respond to system events that may corrupt their state

This is the most concise definition of the requirements for Central Services. Most of this already exists in the various drivers for various platforms. Central Services provides access to that functionality and coordinates between the various APIs that may wish to do that at the same time.

Note that in the descriptions below, the above functions are condensed in cases where they overlap (memory allocation, for example).

### 1.4.1 Provide Information about the System Configuration

*Central Services* provides client APIs with all information they need to return to their client programs and that they may need to make decisions that are dependent upon configuration. This includes:

- Revision of Central Services protocol
- Type of device
- Amount of memory available on the device
- Information regarding the revision of the device which may be necessary for bug workarounds, etc.

*Central Services does not provide any information that could be deduced by the client from the information listed above.*

### 1.4.2 Provide Memory Management for All Types of Surfaces

Client APIs and the driver APIs use Central Services for allocating memory for desktops render buffers font/bitmap caches, textures, and command buffers. This ensures that no API will have memory conflicts with another. In some cases, the memory is typed—meaning it is either tiled or liner, etc. In these cases (where legal), Central Services can be requested to return memory from a particular pool or which has been marked as a particular type via a mechanism such as content-addressable memory (CAM).

### 1.4.3 Initialize Hardware for Rendering

In many cases, the hardware will already be initialized by the system BIOS or by the display driver. Thus, no further initialization is necessary. However, full-screen applications may wish to enable more exotic features such as Scan-Line Interleaving. *Central Services* provides the mechanism to do this. ***Note that assertion of default 3D state is not a service that Central Services provides.***

### 1.4.4 Initialize Command Transport to Enable Rendering

Client APIs use *Central Services* to allocate space for storing commands, initialize whatever hardware state must be initialized for using that space.

### 1.4.5 Initialize Video for Rendering

In many cases (such as OpenGL on W9x/NT), the video mode is already set to what the application wants. However, on certain operating systems, triggers must be cocked so that the driver may recover if the application wishes to run full-screen and later loses it through a catastrophic system event or task switch. An example is the case where the CS client needs to use the video overlay registers, but does not need to change the depth, resolution, or refresh rate (video mode) from it's current state. A task switch or other event could cause Windows to revert to the desktop. In this case, Windows would (erroneously) conclude that the video hardware needed no programming in order for the desktop to be displayed. The result is that the user perceives a hang. It is the responsibility of *Central Services* to set any triggers and ensure the application is running with the desired video mode.

### 1.4.6 Get Rendered Pixels to the Screen

In the windowed case, there are two safe ways of getting rendered pixels to the screen: video overlays and blits. Video overlays must be managed, as they are a scarce resource. *Central Services* provides a way for *any* API to use them, and tracks their availability. In order for blits to occur properly, they must be done by a subsystem that has access to a given window's clip list. Thus, *Central Services* provides functionality which enables the client app to request that a surface be blitted to a window with the proper clipping.

### 1.4.7 Command Execution

*Central Services* provides the client application with non-volatile space to place commands. This means that mode changes, task switches, other APIs, etc. will not corrupt the space given to the client. It is Client's responsibility to divide that space up into sections for sequential use.

In the case of state corruption (another API has run since the last time the CS client has executed commands), *Central Services* informs the client (via a return status) that it needs to restore its state. Further CS provides information to the client regarding whether register or memory state has been corrupted so the client can know what is required to continue.

*Central Services* provides the client with another non-volatile place *to which the hardware can render* for use as a sentinel buffer. The client is then responsible for generating command packets that write a serial number to this buffer for use in determining which subdivisions of the command area are free for reuse.

### 1.4.8 Reading Hardware Registers

If any hardware registers must be read during normal operation, it is the responsibility of Central Services to provide that capability to the client API. This bottlenecks reads and insures they do not interfere with the operation of any other APIs. However, given the nature of the cooperation between driver and non-driver APIs, functions like idle, etc. are performed by reading the sentinel buffers described above.

# 2 Functional Overview

*Central Services* is simply a unified method for gaining access to the hardware and for informing one subsystem that another has caused a situation that requires reaction (such as losing a surface or context). In order to prevent race conditions that can arise when non-kernel APIs are sharing the hardware with kernel APIs, certain types of access must be synchronized. In the *Central Services* model, the display driver owns the hardware. All code that allocates memory, initialize hardware, programs command transport, or performs video programming lives in the display driver. Central Services simply provides a non-kernel API with the ability to use the same code the kernel APIs use and provides a level of coordination between the kernel and non-kernel APIs.

This section states how Central Services will address the requirements listed above by describing the protocol and APIs used to perform operations. Each entry in this section will describe both the interface that the client uses, and the interface that the *Central Services Client Library* uses to communicate with the *Central Services Server*. The Central Services Client Library packs the information that the Client provides into *transport structures*, before sending them to the server via a system-dependent communications channel. On Windows platforms, it is transported between the Central Services client and display driver using the *ExtEscape()* function. On Linux, the channel is an X extension, and on the Mac, some voodoo weirdness occurs.

## 2.1 csInit

Initialize the *Central Services* client library. This must be called once for each application. This function is not process or thread specific.

```
CSRESULT FX_CALL csInit( PCSINIT psInit )
```

- ***psInit**: The memory address of a CSINIT structure. This structure is OS-specific.*

| Result Code (CSRESULT) | Condition |
|---|---|
| CS_SUCCESS | *Successful outcome* |
| CSINIT_APIERROR_ALREADYINITIALIZED | *The system has already been initialized for client associated with the given CSINIT structure..* |
| CSINIT_APIERROR_DIRECTDRAWFAILED | *(Win32) could not initialize DirectDraw* |

## 2.2 csUnInit

Un-Initialize the *Central Services* client library. For now, this function is just used for bookkeeping.

```
CSRESULT FX_CALL csUnInit( FxVOID )
```

| Result Code (CSRESULT) | Condition |
|---|---|
| CS_SUCCESS | *Successful outcome* |
| CS_APIERROR_SYSTEMNOTINITIALIZED | *The system was never initialized with* csInit*()* |

## 2.3 csGetProtocolRevision

Returns the major and minor revision numbers of the *Central Services* protocol, as supported by the server. This request is used by the *Central Services* client to determine if there is sufficient compatibility between the client and the client-library/server pair, in order to proceed.

```
CSRESULT FX_CALL csGetProtocolRevision(
                                        CSWINDOWID idWindow,
                                        FxU32* pu32Major,
                                        FxU32* pu32Minor );
```

- ***idWindow:*** The window ID of the Central Services client. This datatype is OS-specific. (For WIN32 clients, this was passed to the client as the first parameter of its registered window message handling procedure.)
- ***pu32Major:*** The memory address at which the server should store the major revision of the protocol (CS_PROTOCOL_MAJOR). If the major revision of the client and the display driver do not match, no functionality is guaranteed.
- ***pu32Minor:*** The memory address at which the server should store the minor revision of the protocol (CS_PROTOCOL_MINOR). If the minor revision of the server is greater than or equal to the minor revision of the client, then all functionality known to the client is guaranteed to work (assuming the major revisions match.) If the minor revision of the server is less than the minor revision of the client, then there is not sufficient compatibility to run.

***Note:*** Generally, the client-library is thought of as a "dumb pipe" that abstracts the OS-specifics of connecting to the *Central Services* server. However, in some cases, some of the server-side implementation may actually be implemented in the Client Library, rather than in the Server itself. (A notable examples would be the fact that portions of csSwapBufferToDisplay() are implemented in the client-library.) When this occurs, the client-library must qualify itself against the server, *in addition* to the normal activity of passing this information on to the client, so that the client can do the same.

| **Result Code (CSRESULT)** | **Condition** |
|---|---|
| CS_SUCCESS | *Successful outcome* |
| CS_APIERROR_INVALIDPARAM | *An invalid parameter was passed in.* |
| CS_APIERROR_SYSTEMNOTINITIALIZED | *The system was never initialized with csInit()* |
| CS_APIERROR_SERVERFAILEDEXTESCAPE | *The server returned a server-specific error code.* |
| CS_APIERROR_SYSTEMFAILEDEXTESCAPE | *(WIN32) The ExtEscape() system call failed* |

## 2.4 csGetGraphicalContext

Returns information about the physical configuration of the device on which the Central Services client will run. Central Services uses the supplied window ID to identify the appropriate device.

```
CSRESULT FX_CALL csGetGraphicalContext(
                                        CSWINDOWID idWindow,
                                        PCSGRAPHICALCONTEXT psGraphicalContext );
```

- ***idWindow:*** The window ID of the Central Services client. For win32 clients, this was passed to the client as the first parameter of its registered window message handling procedure.
- ***psGraphicalContext:*** The memory address of a client-supplied *CSGRAPHICALCONTEXT* structure, to be filled in by the server and/or client library. The client will use this as a handle when calling all higher-level Central Services functions.

| **Result Code (CSRESULT)** | **Condition** |
|---|---|
| CS_SUCCESS | *Successful outcome* |
| CS_APIERROR_INVALIDPARAM | *An invalid parameter was passed in.* |
| CS_APIERROR_SYSTEMNOTINITIALIZED | *The system was never initialized with csInit()* |
| CS_APIERROR_SERVERFAILEDEXTESCAPE | *The server returned a server-specific error code.* |
| CS_APIERROR_SYSTEMFAILEDEXTESCAPE | *(WIN32) The ExtEscape() system call failed* |
| CS_APIERROR_OUTOFMEMORY | *The server could not obtain enough system memory to create and initialize the data structures needed to support this new context.* |
| CS_APIERROR_INVALIDCHIPTYPE | *The server returned a chip type that the client library didn't recognize.* |
| CS_APIERROR_GRAPHICALCONTEXTINITIALIZED | *The CSGRAPHICALCONTEXT structure referenced by psGraphicalContext appears to already be initialized.* |
| CSGETGRAPHICALCONTEXT_APIERROR_IDWINDOWASSIGNEDTOGRAPHICALCONTEXT | *Only one graphical context is allowed per each CSWINDOWID.* |
| CSGETGRAPHICALCONTEXT_APIERROR_CANTCREATECLIPPER | *(WIN32) DirectDraw wouldn't return a 'this' pointer to a Clipper object.* |
| CSGETGRAPHICALCONTEXT_APIERROR_CANTASSOCIATEWINDOWTOCLIP | *(WIN32) DirectDraw reported an error when attempting to associate 'idWindow' with the Clipper object* |

## 2.5 csRegisterStateCallback

(rvb)

Occasionally, events outside a client's control will cause it to lose (or gain) access to the hardware. *Central Services* monitors the system for these type of events, and uses a client-supplied callback function to inform clients of changes in their status.

```
CSRESULT FX_CALL csRegisterStateCallback(
                                        CSWINDOWID idWindow,
                                        PCSCALLBACK pExclusiveCallback );
```

- ***idWindow:*** The window ID of the Central Services client. For win32 clients, this was passed to the client as the first parameter of its registered window message handling procedure.
- ***pExclusiveCallback:*** The memory address of a client-supplied callback function that conforms to the PCSCALLBACK function datatype.

***Note:*** Clients can't monitor this for themselves, because not only does a client have to be notified that it's gained exclusivity, the previously exclusive app has to be told that it lost it. Due to the asynchronous nature of window messages under Win9x, it cannot be guaranteed that the loser will be notified before the new owner (which is essential.) This means that exclusivity checking fails the Software Litmus Test, and therefore must be handled by *Central Services* (which has enough state information to notify the loser first, even if it receives the window messages in the wrong order.)

## 2.6 csAlloc

Allocates a memory buffer of the specified type and dimensions. This request is used to allocate any type of graphics memory a that would normally be used by a display driver. (Buffer type and memory type are separated because certain types of buffers can be in either tiled or linear memory in the frame buffer, or in AGP memory on the system.) Note that—in general—the server and client library make no attempt to enforce the designated use of this buffer. It is considered perfectly legal for the client to use the buffer for a purpose other than the intent it declared at **csAlloc** time. ***Exception: on SST2, a surface allocated with 'u32UseSLI' set (nonzero) must NOT be used for non-SLI rendering, and vice versa***. Buffers can be placed in either local frame buffer memory or AGP memory. However, on some hardware, rendering to AGP is not supported, thus this operation can fail. The Central Services client should use the device ID (obtained from) to determine if the part is capable of rendering to-- or texturing from-- AGP memory.

```
CSRESULT FX_CALL csAlloc(
                        PCSGRAPHICALCONTEXT     psGraphicalContext,
                        PCSALLOCATIONDESCRIPTOR psAllocationDescriptor );
```

- ***psGraphicalContext:*** The memory address of a CSGRAPHICALCONTEXT structure, that serves as a context handle. This handle was initialized and returned to the client when it called csGetGraphicalContext().
- ***psAllocationDescriptor:*** The memory address of a client-supplied CSALLOCATIONDESCRIPTOR structure. Some of the fields in this structure will be initialized by the client prior to calling this function; others will be initialized by the server, as a result of the call.

| **Result Code (CSRESULT)** | **Condition** |
|---|---|
| CS_SUCCESS | *Successful outcome* |
| CS_APIERROR_INVALIDPARAM | *An invalid parameter was passed in.* |
| CS_APIERROR_SYSTEMNOTINITIALIZED | *The system was never initialized with csInit()* |
| CS_APIERROR_SERVERFAILEDEXTESCAPE | *The server returned a server-specific error code.* |
| CS_APIERROR_SYSTEMFAILEDEXTESCAPE | *(WIN32) The ExtEscape() system call failed* |
| CS_APIERROR_INVALIDCONTEXT | *The context ID passed in is not that of a valid context.* |
| CS_APIERROR_INVALIDPARAM | *An invalid parameter was passed in.* |
| CS_APIERROR_OUTOFMEMORY | *The server could not obtain enough system memory to create and initialize the data structures needed to support this new buffer, or the server could not find a chunk of video memory big enough to satisfy the allocation request.* |
| CS_APIERROR_ALLOCATIONDESCRIPTORINITIALIZED | *The allocation descriptor passed in by the client appears to already be initialized for another valid surface.* |
| CSALLOC_APIERROR_INVALIDCOMB | *Invalid combination of buffer type and memory type.* |
| CSALLOC_APIERROR_INVALIDSIZE | *The buffer size requested is not supported with the buffer and memory types that were specified.* |
| CSALLOC_APIERROR_INVALIDMEMORYTYPE | *The 'u32MemType' field has an invalid or unsupported value.* |
| CSALLOC_APIERROR_INVALIDLOCALE | *The 'u32Locale' field has an invalid or unsupported value.* |
| CSALLOC_APIERROR_INVALIDBUFFERTYPE | *The 'u32BufferType' field has an invalid or unsupported value.* |

## 2.7 csFree

Frees a buffer, previously allocated with csAlloc() .

```
CSRESULT FX_CALL csFree(
                        PCSGRAPHICALCONTEXT     psGraphicalContext,
                        PCSALLOCATIONDESCRIPTOR psAllocationDescriptor );
```

- ***psGraphicalContext:*** The memory address of a CSGRAPHICALCONTEXT structure, that serves as a context handle. This handle was initialized and returned to the client when it called csGetGraphicalContext()
- ***psAllocationDescriptor:*** The memory address of a client-supplied CSALLOCATIONDESCRIPTOR structure, previously initialized by csAlloc().

| **Result Code (CSRESULT)** | **Condition** |
|---|---|
| CS_SUCCESS | *Successful outcome* |
| CS_APIERROR_INVALIDPARAM | *An invalid parameter was passed in.* |
| CS_APIERROR_SYSTEMNOTINITIALIZED | *The system was never initialized with csInit()* |
| CS_APIERROR_SERVERFAILEDEXTESCAPE | *The server returned a server-specific error code.* |
| CS_APIERROR_SYSTEMFAILEDEXTESCAPE | *(WIN32) The ExtEscape() system call failed* |
| CSFREE_APIERROR_STILLLOCKED | *The client attempt to free a surface for a lock was active.* |

## 2.8 csSetVideoMode

(rvb)

*Central Services* clients use **csSetVideoMode** function sets the video mode as specified by its arguments. If the mode is not supported by the display driver/graphics device/monitor combination, the mode will not be set. Further, the requested refresh rate is merely a suggestion on Windows platforms, as WHQL dictates that the driver must obey the settings in the video control panel. *Note that only a client running in exclusive mode may set the video mode.*

```
CSRESULT FX_CALL csSetVideoMode(
                                PCSGRAPHICALCONTEXT psGraphicalContext,
                                PCSVIDEOMODE        psVideoMode );
```

***psGraphicalContext:*** The memory address of a CSGRAPHICALCONTEXT structure, that serves as a context handle. This handle was initialized and returned to the client when it called.

- ***psVideoMode:*** The memory address of a CSVIDEOMODE structure, containing a description of the desired video mode.

| **Result Code (CSRESULT)** | **Condition** |
|---|---|
| CS_SUCCESS | *Successful outcome* |
| CS_APIERROR_INVALIDPARAM | *An invalid parameter was passed in.* |
| CS_APIERROR_SYSTEMNOTINITIALIZED | *The system was never initialized with csInit()* |
| CS_APIERROR_SERVERFAILEDEXTESCAPE | *The server returned a server-specific error code.* |
| CS_APIERROR_SYSTEMFAILEDEXTESCAPE | *(WIN32) The ExtEscape() system call failed* |
| CSSETVIDEOMODE_APIERROR_UNSUPPORTEDRES | *Either there is not enough frame buffer memory to support the requested resolution, or the combination of width, height, and color depth requested are not supported by the display adapter/monitor/driver combination.* |
| CSSETVIDEOMODE_STATUS_BADREFRESH | *The video mode was changed to the requested resolution, but at a different refresh rate than the one requested.* |
| CSSETVIDEOMODE_APIERROR_NOTFULLSCREEN | *The context associated with the request is not full-screen* |
| CSVIDINIT_ERROR_UNSUPPORTEDRES | *The width and height are not supported by the display adapter/monitor/driver combination.* |
| CSSETVIDEOMODE_APIERROR_AAUNSUPPORTED | *The hardware in question does not support antialiasing* |

## 2.9 csAcquireOverlay

*Central Services* clients use **csAcquireOverlay** (rvb) to obtain ownership of an overlay hardware resource, if one is available. An error will result if all hardware overlay resources are in use, and the calling CS client is not running in exclusive mode. ***Server specific:*** *If the calling CS client is in exclusive mode, and all available overlays are in use, then an overlay* ***may*** *be taken away from another app to satisfy the needs of the calling CS client. But this is entirely the server's call.*

Within the context of *Central Services*, overlays are useful for:

- multi-buffering to a window (underlay)
- displaying video clips (underlay)
- imposing a mask over a rendered image (overlay)

```
CSRESULT FX_CALL csAcquireOverlay(
                                 PCSGRAPHICALCONTEXT psGraphicalContext,
                                 PCSOVERLAY          psOverlay );
```

- ***psGraphicalContext:*** The memory address of a CSGRAPHICALCONTEXT structure, that serves as a context handle. This handle was initialized and returned to the client when it called.
- ***psOverlay:*** The memory address of a CSOVERLAY structure, used to describe the acquired overlay.

| **Result Code (CSRESULT)** | **Condition** |
|---|---|
| CS_SUCCESS | *Successful outcome* |
| CS_APIERROR_INVALIDPARAM | *An invalid parameter was passed in.* |
| CS_APIERROR_SYSTEMNOTINITIALIZED | *The system was never initialized with csInit()* |
| CS_APIERROR_SERVERFAILEDEXTESCAPE | *The server returned a server-specific error code.* |
| CS_APIERROR_SYSTEMFAILEDEXTESCAPE | *(WIN32) The ExtEscape() system call failed* |
| CSACQUIREOVERLAY_APIERROR_UNSUPPORTED | *The hardware has no overlay support* |
| CSACQUIREOVERLAY_APIERROR_OVERLAYBUSY | *Another client or process has ownership of the overlay* |

## 2.10 csSwapBufferToDisplay

*Central Services* clients use **csSwapBufferToDisplay** to copy data from a CS render buffer to the screen, when no overlay is available. To this end, **csSwapBufferToDisplay** will use the blit engine of the hardware to blit the offscreen render target onto the window, honoring any clip information that the OS provides.

As this solution is obviously not optimal, the client should first determine if a hardware overlay resource is available (via csAcquireOverlay(),) and if so, use it instead. (rvb)

When the source rectangle and the destination rectangle are different sizes, **csSwapBufferToDisplay** will either shrink or stretch the data to fit—using the differences in size to determine the scaling factor(s). If either stretch or shrink is indicated, the *interpolationType* variable is used to specify whether point sampling or bilinear interpolation is used. If the extents of the destination rectangle reach beyond the destination window, the blit is clipped. Furthermore, the blit is clipped as indicated by the display driver's clip list. **csSwapBufferToDisplay** will also attempt to color convert, if necessary and possible.

```
CSRESULT FX_CALL csSwapBufferToDisplay(
                                    PCSGRAPHICALCONTEXT    psGraphicalContext,
                                    PCSSWAPBUFFERTODISPLA  SwapBufferToDisplay )
```

- ***psGraphicalContext:*** The memory address of a CSGRAPHICALCONTEXT structure, that serves as a context handle. This handle was initialized and returned to the client when it called.
- ***psSwapBufferToDisplay:*** The memory address of a CSSWAPBUFFERTODISPLAY structure, containing information about the source and destination surfaces.

| **Result Code (CSRESULT)** | **Condition** |
|---|---|
| CS_SUCCESS | *Successful outcome* |
| CS_APIERROR_INVALIDPARAM | *An invalid parameter was passed in.* |
| CS_APIERROR_SYSTEMNOTINITIALIZED | *The system was never initialized with csInit()* |
| CS_APIERROR_SERVERFAILEDEXTESCAPE | *The server returned a server-specific error code.* |
| CS_APIERROR_SYSTEMFAILEDEXTESCAPE | *(WIN32) The ExtEscape() system call failed* |
| CS_APIERROR_INVALIDWINDOW | *The window ID passed in by the client is not valid.* |
| CS_APIERROR_SURFACELOST | *One or more of the source and/or destination surfaces were lost, as a result of a full-screen context becoming active, or the surface getting evicted by a mode switch.* |
| CSSWAPBUFFERTODISPLAY_APIERROR_UNSUPPORTEDCONV | *The specified color-space conversion is unsupported by hardware.* |
| CSSWAPBUFFERTODISPLAY_APIERROR_UNSUPPORTEDSCALE | *The implied scale factor is outside of the range supported by the hardware.* |
| CSSWAPBUFFERTODISPLAY_APIERROR_CANTGETCLIPLISTSIZE | *The OS call to obtain the size of the destination clip list failed.* |
| CSSWAPBUFFERTODISPLAY_APIERROR_CANTGETCLIPLIST | *The OS call to obtain the destination clip list failed.* |

## 2.11 csExecuteCommands

(rvb)

The **csExecuteCommands** function causes a command buffer to be executed.

```
CSRESULT FX_CALL csExecuteCommands(
                 PCSGRAPHICALCONTEXT     psGraphicalContext,
                 PCSUCODESTATEDESCRIPTOR psUCodeStateDescriptor,
                 PCSALLOCATIONDESCRIPTOR psStateAllocationDescriptor,
                 PCSALLOCATIONDESCRIPTOR psCommandAllocationDescriptor,
                 FxU32                   u32UCodeSize,
                 FxU32                   u32StateSize,
                 FxU32                   u32CommandSize,
                 FxU32                   u32StateOffset,
                 FxU32                   u32CommandOffset );
```

- ***psGraphicalContext:*** The memory address of a CSGRAPHICALCONTEXT structure, that serves as a context handle. This handle was initialized and returned to the client when it called.
- ***psUCodeStateDescriptor:*** The memory address of a buffer that contains a description of the expected microcode state. (rvb)[rvb9]
- ***psStateAllocationDescriptor:*** The Allocation descriptor of the buffer of commands that would restore the hardware state, if necessary.
- ***psCommandAllocationDescriptor:*** Allocation descriptor of the buffer of commands to be executed.
- ***u32UCodeSize:*** The total size of the psUCodeStateDescriptor buffer, in bytes.
- ***u32StateSize:*** The total size of the psStateAllocationDescriptor buffer, in bytes.
- ***u32CommandSize:*** The total size of the psCommandAllocationDescriptor buffer, in bytes.
- ***u32StateOffset:*** The offset inside psStateAllocationDescriptor, that marks the beginning of the state information.
- ***u32CommandOffset:*** The offset inside psCommandAllocationDescriptor, that marks where the commands are stored.

| **Result Code (CSRESULT)** | **Condition** |
|---|---|
| CS_SUCCESS | Successful outcome |
| CS_APIERROR_SYSTEMNOTINITIALIZED | *The system was never initialized with csInit()* |
| CS_APIERROR_INVALIDCONTEXT | *The context ID passed in is not that of a valid context.* |
| CS_APIERROR_SURFACELOST | *One or more of the relevant surfaces were lost, as a result of a full-screen context becoming active, or the surface getting evicted.* |
| CS_APIERROR_FULLSCREENACTIVE | *A full-screen context is presently active, so buffer execution requests cannot be honored at the moment.* |

## 2.12 csLock

Central Services clients use **csLock** to get an address for use in accessing the buffer they specify as a linear frame buffer. If both **CSLOCK_FLAG_READABLE** and **CSLOCK_FLAG_WRITEABLE** are specified, a read/write lock is indicated. Certain combinations of lock type and 3D aperture may not be supported (some platforms do not support reads through the 3D LFB aperture). Further, if the device has a CAM for LFB access, the lock can fail due to a dearth of CAM entries.

**Rampage Implementation Note:** (rvb)

```
CSRESULT FX_CALL csLock(
                    PCSGRAPHICALCONTEXT     psGraphicalContext,
                    PCSALLOCATIONDESCRIPTOR psAllocationDescriptor );
```

- ***psGraphicalContext:*** The memory address of a CSGRAPHICALCONTEXT structure, that serves as a context handle. This handle was initialized and returned to the client when it called.
- ***psAllocationDescriptor:*** The memory address of a client-supplied CSALLOCATIONDESCRIPTOR structure. Some of the fields in this structure will be initialized by the client prior to calling this function; others will be initialized by the server.

| Result Code (CSRESULT) | Condition |
|---|---|
| CS_SUCCESS | *Successful outcome* |
| CS_APIERROR_SYSTEMNOTINITIALIZED | *The system was never initialized with csInit()* |
| CS_APIERROR_INVALIDCONTEXT | *The context ID passed in is not that of a valid context.* |
| CS_APIERROR_SURFACELOST | *One or more of the relevant surfaces were lost, as a result of a full-screen context becoming active, or the surface getting evicted.* |
| CS_APIERROR_FULLSCREENACTIVE | *A full-screen context (which has not lost its surfaces) is presently active, so a new context cannot be created.* |
| CSLOCK_APIERROR_UNSUPPORTEDLOCK | *Bad combination of LFB aperture and lock type* |
| CSLOCK_APIERROR_NOLOCKAVAILABLE | *The request failed due to exceeding the lock count. This is usually due to some physical limitation, such as no free CAM available to instantiate the lock.* |

## 2.13 csUnlock

Central Services clients use **csUnlock** to inform the driver that they no longer need LFB access to the specified buffer. This is necessary because on some architectures, limited resources (such as CAMs) can be associated with LFB access. It is considered poor form to lock a buffer and leave it locked, because there can be limited resources (like CAM entries) associated with LFB access to certain types of memory.

```
CSRESULT FX_CALL csUnlock(
                        PCSGRAPHICALCONTEXT psGraphicalContext,
                        PCSALLOCATIONDESCRIPTOR psAllocationDescriptor );
```

- ***psGraphicalContext:*** The memory address of a CSGRAPHICALCONTEXT structure, that serves as a context handle. This handle was initialized and returned to the client when it called.
- ***psAllocationDescriptor:*** The allocation descriptor returned by csAlloc.

| **Result Code (CSRESULT)** | **Condition** |
|---|---|
| CS_SUCCESS | *Successful outcome* |
| CS_APIERROR_SYSTEMNOTINITIALIZED | *The system was never initialized with csInit()* |
| CS_APIERROR_INVALIDCONTEXT | *The context ID passed in is not that of a valid context.* |
| CS_APIERROR_SURFACELOST | *One or more of the relevant surfaces were lost, as a result of a full-screen context becoming active, or the surface getting evicted.* |
| CSUNLOCK_APIERROR_NOTLOCKED | *The specified buffer is not locked, so it can't be unlocked.* |

## 2.14 csDEBUGGetStringForError

This function returns an error string that is associated with the Central Services error returned from each function.

```
PFxSz FX_CALL csDEBUGGetStringForError( CSRESULT idError )
```

The returned string must not be written to!!!

## 2.15 csReleaseGraphicalContext

Central Services clients must call *csReleaseGraphicalContext* when they no longer need the specified context. This allows the driver to release any resources associated with the context.

```
CSRESULT FX_CALL csReleaseGraphicalContext(
                PCSGRAPHICALCONTEXT psGraphicalContext );
```

- ***psGraphicalContext:*** The memory address of a CSGRAPHICALCONTEXT structure, that serves as a context handle. This handle was initialized and returned to the client when it called.

| **Result Code (CSRESULT)** | **Condition** |
|---|---|
| CS_SUCCESS | *Successful outcome* |
| CS_APIERROR_SYSTEMNOTINITIALIZED | *The system was never initialized with csInit()* |
| CS_APIERROR_INVALIDCONTEXT | *The context ID passed in is not that of a valid context.* |

# 3 Data Structures / Data Types

## 3.1 CSINIT

This structure is OS-specific. Below is a description of each field, for each specific operating system.

```
typedef struct csinit_s
{
  /* contents are entirely OS-specific */

} CSINIT, FxFAR *PCSINIT;
```

**WIN32 (Windows 9x, Windows NT, Windows 2K)**

| | |
|---|---|
| `HINSTANCE hDLLInstance` | *The client should initialize this field to the value passed in to the first parameter of the client's "DllMain()" function, prior to calling csInit().* |

## 3.2 CSGRAPHICALCONTEXT

This structure serves as a context handle for the Central Services client. Central Services stores client-specific state information in this structure.

```
#define CSGRAPHICALCONTEXT_FLAGS_INITIALIZED    0x00000001
typedef struct csgraphicalcontext_s
{
    CSSLLISTNODE sNextNodeGraphicalContext
    CSSLLISTNODE sRootNodeAllocationDescriptor;
    FxU32       u32Flags;
    FxU32       u32ContextID;
    CSWINDOWID  idWindow;
    CSPROCESSID idProcess;
    CSTHREADID  idThread;
    CSVIDEOMODE sPreviousVideoMode;
    CSVIDEOMODE sCurrentVideoMode;
    CSOVERLAY   sOverlay;
    CSDEVICECONFIG sDeviceConfig;

#if (defined(WIN32) || (defined(WIN95) && defined(IS_16)))
    CSHDC hDC;
    LPDIRECTDRAWCLIPPER psDirectDrawClipper;
#endif /* WIN32 */

    CSCHIPSPECIFICDATA  sChipSpecificData;
    PCSCALLBACK         pStateCallback;

} CSGRAPHICALCONTEXT, FxFAR*PCSGRAPHICALCONTEXT;
```

- ***sNextNodeGraphicalContext:*** Internal use only.
- ***sRootNodeAllocationDescriptor:*** Internal use only.
- ***u32Flags:*** Internal use only.
- ***u32ContextID:*** Internal use only.
- ***idWindow:*** The handle of the window associated with this graphical context.
- ***idProcess:*** Internal use only.
- ***idThread:*** Internal use only.
- ***sPreviousVideoMode:*** Internal use only
- ***sCurrentVideoMode:*** Internal use only
- ***sOverlay:*** Internal use only (rvb)
- ***sDeviceConfig:*** A structure via which the server provides identification information about the underlying hardware.
- ***sChipSpecificData:*** A structure via which the server provides information about the capabilities of the underlying hardware.
- ***pStateCallback:*** The address of a function that Central Services can call to inform the client of various asynchronous changes to the system state. The provided callback function is expected to match this prototype:
  `void <function-name> (PCSGRAPHICALCONTEXT, FxU32);`

Additionally, for WIN32 there are some OS-specific fields in this data structure.

- ***hDC:*** Internal use only
- ***psDirectDrawClipper:*** a 'this' pointer to the DirectDraw Clipper object that is associated with this client's window.

## 3.3 PCSCALLBACK

```
#define CS_STATE_SLIPOSSIBLE      0x00000001
#define CS_STATE_SLINOTPOSSIBLE   0x00000002
#define CS_STATE_EXCLUSIVE        0x00000003
#define CS_STATE_NOTEXCLUSIVE     0x00000004
#define CS_STATE_OVERLAYAVAILABLE 0x00000005
#define CS_STATE_OVERLAYLOST      0x00000006

typedef void (*PCSCALLBACK)( PCSGRAPHICALCONTEXT psGraphicalContext,
                             FxU32 u32Message, FxVOID* param );
```

## 3.4 CSCHIPSPECIFICDATA

This structure is used for multiple purposes, and in multiple contexts. (That's context as in conceptual, not context as in graphical.) This structure is used to:

- Report device capabilities ()
- Report device, resource, or attribute status ()
- Change device, resource, or attribute status ()

```
/* Chip types for the chip specific data field */
#define CSCHIPSPECIFICDATA_CHIPTYPE_SST2  0x01

typedef struct cschipspecificdata_s
{
   FxU32   u32Flags              /* Undefined */
   FxU32   u32ChipType;          /* Chip type for this structure.
                                    CSCHIPSPECIFICDATA_CHIPTYPE_SST2
                                 */
    union
    {
      CSSST2 sSST2;              /* sst2 chip specific data */

    } unionChipType;

} CSCHIPSPECIFICDATA, *PCSCHIPSPECIFICDATA;
```

- ***u32Flags:*** A field for non-chip specific flags in a chip-specific data structure.
- ***u32ChipType:*** A field that identifies the rendering hardware exposed by the Central Services server.
- ***unionChipType:*** A union of structures, each representing a specific supported chip, such as for SST2.

## 3.5 CSSST2

This structure is part of the *unionChipType* union in CSCHIPSPECIFICDATA. The meaning of its various fields will vary from one context to another (That's context as in conceptual, not context as in graphical.)

```
/* Chip specific data for the sst2 chipset */
#define CSSST2_REGSPACE_BASE  0x01
#define CSSST2_REGSPACE_IO    ( CSSST2_REGSPACE_BASE + 0x00 )
#define CSSST2_REGSPACE_2D    ( CSSST2_REGSPACE_BASE + 0x01 )
#define CSSST2_REGSPACE_3D    ( CSSST2_REGSPACE_BASE + 0x02 )
#define CSSST2_REGSPACE_CMD   ( CSSST2_REGSPACE_BASE + 0x03 )
#define CSSST2_REGSPACE_PCI   ( CSSST2_REGSPACE_BASE + 0x04 )

typedef struct cssst2_s
{
    FxU32   u32Flags;
    FxU32   u32SLIAvailable;      (rvb)
    FxU32   u32NumSLIChips;       (rvb)
    FxU32   u32UseSLI;            (rvb)
    FxU32   u32DeviceRev;
    FxU32   u32LFBRam;            (rvb)
    FxU32   u32AGPRam;            (rvb)
    FxU32   u32Antialiased;       (rvb)
    FxU32   u32CAMEntry;
    FxU32   u32TileMode;
    FxU32   u32TileWidth;
    FxU32   u32TileHeight;
    FxU32   u32RegisterSpace;     /* CSSST2_REGSPACE_XXXXYYYYY */
    FxU32   u32RegisterOffset;
    FxU32   u32IOSize;
    FxU32   u32RegisterValue;
    PFxVOID pvIOBase;
    PFxVOID pvMemBase0;
    PFxVOID pvMemBase1;

} CSSST2,FxFAR*PCSSST2;
```

- ***u32Flags:*** A field for chip-specific flags. Presently not used.
- ***u32SLIAvailable:*** Only used csGetGraphicalContext. Indicates that the underlying hardware is SLI-capable. (see note above.)
- ***u32NumSLIChips:*** The number of chips in the SLI system, if applicable. Can be 0 or 1 if the underlying hardware is not SLI-capable. (see note above.)
- ***u32UseSLI:*** Only used by csSetVideoMode. (extremely deprecated, and may be removed; see note above.)
- ***u32DeviceRev:*** Only used (set) by csGetGraphicalContext. Reports the silicon revision level of the underlying hardware.
- ***u32LFBRam:*** Only used (set) by csGetGraphicalContext. Reports the amount of frame buffer memory attached to the underlying hardware (this value is per-chip, not per-board; so a 4-way SLI board will actually have 4x more LFB memory than what is reported here.) (see note above.)
- ***u32AGPRam:*** Only used (set) by csGetGraphicalContext. Reports the amount of AGP memory available to the underlying hardware. (see note above.)
- ***u32Antialiased:*** When set by csGetGraphicalContext, this field indicates the way-ness of the underlying hardware's AA capability (usually 2-way or 4-way). When

used (set/cleared) by the client prior to calling csAlloc, this field indicates whether the allocated buffer should be allocated as an anti-aliased surface. (see note above.)

- ***u32CAMEntry:*** Only used by csLock (set) and csUnlock (clear). Indicates which CAM entry in SST2 was used for the lock**.**
- ***u32TileMode:*** When used (set) by the client prior to calling csAlloc, this field indicates the appropriate tile mode of the tiled surface that is being allocated. The SST2 server will only read this field if the CS_MEMORY_TILED but was set in the allocation descriptor. When used (read) by csSwapBufferToDisplay, this field tells the SST2 server how to format the packets that instruct WAX to blit the tiled buffer to the screen.
- ***u32TileWidth:*** The width of the tile in *pixels*, as defined by *u32TileMode*.
- ***u32TileHeight:*** The height of the tile, in *pixels*, as defined by *u32TileMode*.
- ***u32RegisterSpace:***
- ***u32RegisterOffset:***
- ***u32IOSize:***
- ***u32RegisterValue:***
- ***pvIOBase:***
- ***pvMemBase0:***
- ***pvMemBase1:*** These 7 fields are used in a csDeviceSpecificCommunication call. Their usage is highly discouraged.

## 3.6 CSDEVICECONFIG

This structure is a substructure of CSGRAPHICALCONTEXT. It provided chip-specific information about the underlying rendering hardware.

```
typedef struct csdevicecontext_s  (rvb)
{
    FxU32 u32Flags;          /* Undefined */
    FxU32 u32ContextID;      /* context ID */
    CSWINDOWID idWindow;     /* window ID */
   CSPROCESSID idProcess;  /* Process ID */
#ifdef WIN32
    HDC hDesktopDC;          /* WIN32 device context for ExtEscape
                                calls */
#endif /* WIN32 */
    CSCHIPSPECIFICDATA  sChipSpecificData;

} CSDEVICECONTEXT, *PCSDEVICECONTEXT;
```

## 3.7 CSALLOCATIONDESCRIPTOR

This structure is used to manage buffers allocated by *Central Services*.

```
#define CSALLOCATIONDESCRIPTOR_FLAGS_INITIALIZED    0x00000001
#define CSALLOCATIONDESCRIPTOR_FLAGS_LOCKED         0x00000002
typedef struct csallocationdescriptor_s
{
    CSSLLISTNODE sNextNodeAllocationDescriptor
    FxU32   u32Flags
    FxU32   u32BufferID;
    FxU32   u32BufferType;
    FxU32   u32MemType;
    FxU32   u32Locale;
    FxU32   u32Size;
    FxU32   u32Width;
    FxU32   u32Height;
    FxU32   u32Depth;
    FxU32   u32PhysicalOffset;
    PFxVOID pvLinearAddress;
    FxU32   u32LockCount;
    FxU32   u32LockFlags;
    FxU32   u32LinearStride;
    FxU32   u32LFBDepth;
    FxU32   u32PhysicalStride;

    CSCHIPSPECIFICDATA  sChipSpecificData;

} CSALLOCATIONDESCRIPTOR, FxFAR* PCSALLOCATIONDESCRIPTOR;
```

- ***u32Flags:*** Flags that allow the client library and server to keep track of the state of this buffer. The flags are listed above.
- ***u32BufferID:*** An ID that uniquely identifies this buffer.
- ***u32BufferType:*** A field that identifies the intended use of the buffer. Valid values are:
  - CS_BUFFER_FIFO
  - CS_BUFFER_PERSISTENT
  - CS_BUFFER_RENDER
  - CS_BUFFER_TEXTURE
  - CS_BUFFER_ZBUFFER
- ***u32MemType:*** A field that identifies the intended surface flavor of the buffer. Valid values are:
  - CS_MEMORY_LINEAR
  - CS_MEMORY_TILED
- ***u32Locale:*** A field that indicates the intended locale of the buffer. Valid values are:
  - CS_LOCALE_FRAMEBUFFER
  - CS_LOCALE_AGP
- ***u32Size:*** For *linear* surfaces, this is set by the *client* to indicate the *desired* size of the buffer. For *tiled* surfaces, this is set by the *server* to indicate the *actual* size of the buffer.
- ***u32Width:*** (Only applies to tiled surfaces.) The desired width of the surface, in *pixels*.
- ***u32Height:*** (Only applies to tiled surfaces.) The desired height of the surface, in *pixels*.

- ***u32Depth:*** For tiled surfaces (and linear, when appropriate) this indicates the client's intended color depth of the buffer, in *bits*. For linear buffers for which a color depth doesn't apply, this should be set to 8 by the client.
- ***u32PhysicalOffset:*** This field is initialized by the server, and indicates the offset of the allocated buffer, in the locale it was allocated in (AGP or frame buffer.)
- ***pvLinearAddress:*** This field is only valid after a successful call to csLock, and becomes invalid after a call to csUnlock. It indicates the host address that maps to the physical offset of the buffer.
- ***u32LockCount:*** The number of outstanding locks on this buffer.
- ***u32LockFlags:*** Flags that indicate the lock's status and attributes. Available flags are:
  - CSLOCK_FLAG_READABLE
  - CSLOCK_FLAG_WRITEABLE
  - CSLOCK_FLAG_3DLFB
- ***u32LinearStride:*** This field is only valid after a successful call to csLock, and becomes invalid after a call to csUnlock. It indicates the apparent stride of a tiled surface, in *bytes*.
- ***u32LFBDepth:*** This field is initialized by the client, and indicates the LFB depth of the surface. The server ignores this field unless the client set the CSLOCK_FLAG_3DLFB flag, prior to calling csLock.
- ***u32PhysicalStride:*** This field is initialized by the server to indicate the actual stride of the tiled surface, in *bytes*. In some contexts, the client may need to convert this value to # of tiles before using. See the *u32TileWidth* and *u32TileHeight* fields in the CSCHIPSPECIFICDATA structure.

## 3.8 CSVIDEOMODE

(rvb)

```
typedef struct csvideomode_s
{
    FxU32  u32Flags;
    FxU32  u32Width;
    FxU32  u32Height;
    FxU32  u32ColorFormat;
    FxU32  u32RefreshRate;
    CSCHIPSPECIFICDATA sChipSpecificData;

} CSVIDEOMODE, FxFAR* PCSVIDEOMODE;
```

## 3.9 CSOVERLAY

(RVB)

```
#define CSOVERLAY_FLAGS_INITIALIZED         0x00000001
typedef struct csoverlay_s
{
    FxU32 u32Flags;
    FxU32 u32ColorFormat;
    FxU32 u32SrcWidth;
    FxU32 u32SrcHeight;
    FxU32 u32DestX;
    FxU32 u32DestY;
    FxU32 u32DestWidth;
    FxU32 u32DestHeight;
    CSCHIPSPECIFICDATA sChipSpecificData;
} CSOVERLAY, FxFAR*PCSOVERLAY;
```

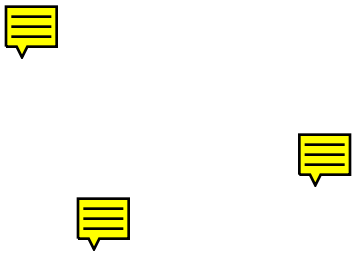
## 3.10 CSSWAPBUFFERTODISPLAY

This structure is used when the client needs the server to blit an offscreen rendered buffer to an onscreen location, while honoring an OS-supplied cliplist.

```
typedef struct csswapbuffertodisplay_s
{
    FxU32                   u32Flags
    PCSALLOCATIONDESCRIPTOR psSrcBufferAllocationDescriptor;
    CSRECT                  sDestClipRegion;
    FxU32                   u32DestX;
    FxU32                   u32DestY;
    CSRECT                  sSrcClipRegion;
    FxU32                   u32SrcColorFormat;
    FxU32                   u32InterpolationType;
    CSWINDOWID              idDestWindow;
    CSCLIPLIST              sClipList;

} CSSWAPBUFFERTODISPLAY,FxFAR*PCSSWAPBUFFERTODISPLAY;
```

- ***u32Flags:*** Reserved
- ***psSrcBufferAllocationDescriptor:*** Internal use only
- ***sDestClipRegion:*** A rectangle that describes the location and size of the destination rectangle, which the client library initialized to correspond to the position and size of the target window.
- ***u32DestX:*** Not sure what this is for. Seems redundant.
- ***u32DestY:*** Not sure what this is for. Seems redundant
- ***sSrcClipRegion:*** This is a “meta-clip” rectangle that serves as a mask against the clip list (rvb)
- ***u32SrcColorFormat:*** The server compares this to the color format of the primary surface, to determine of colorspace conversion is necessary.
- ***u32InterpolationType:*** The interpolation method to use if the sizes of the source and destination rectangles don’t match. (rvb)
- ***idDestWindow:*** The OS or windowing system-supplied window handle that the clip list applies to. (rvb)
- ***sClipList:*** Clip list information from the OS or windowing system.



# 4 Appendices

## 4.1 FAQ

### 4.1.1 Why must a CS client still use csExecuteCommands when in Exclusive Mode? Isn't it safe to write directly to the cmdfifo registers in that situation?

***Ryan:*** Unfortunately, no. The issue is, some OpenGL games use GDI for their menuing system, even when they've made themselves full-screen. GDI itself uses the command fifo, but it is not a *Central Services* app, so it doesn't play by the same rules. So *csExecuteCommands()* is still required, even when in exclusive mode.

The ideal situation would be that all driver components use *Central Services*, so that this wouldn't be an issue. But *Central Services* is intended to be a short-term solution to a fundamental shortcoming of our driver model. It is expected that the new driver architecture for GP-x will address these same issues in a more holistic and complete manner. But with *Central Services*, GDI always has write access to the hardware, and does not have to play by the same rules.

### 4.1.2 Why does the CS client library and/or server trap my CS client's window messages?

***Ryan:*** Some window messages imply a context switch of one sort or another. An example is alt-tab, which can cause one CS app to lose exclusivity, and another to gain it. The problem is that these window messages are asynchronous; there's no guarantee that the loser will be notified before the winner. This is bad news, because it means that two clients could end up thinking that they own the same hardware resources, if even for a short amount of time. But if the library/server traps all messages for all windows of all the clients, it can notify the loser (via the callback) first, even if it happens to receive the winner's notification first.

Another way this can be handled is for the client to trap it's own focus messages, and then repeatedly ask the server for the desired resource, until it gets it. However, the callback solution has the advantage that polling for the resource (which could require an ExtEscape call) is not required.

### 4.1.3 Do I have to call csSwapBufferToDisplay if I have an overlay resource, and am using it to display my rendered buffers?

***Ryan:*** No. Just update the "start of overlay source" offset register when it's time to swap buffers. (IMPORTANT: See question 4.1.5.) When your client doesn't have exclusivity, the CS client library will monitor window move messages, and keep the destination rectangle lines up with the target window. The only other responsibility you have is to stop using the overlay if you receive "CS_STATE_OVERLAYLOST" via the state callback function.

### 4.1.4 Is the callback necessary?

***Ryan:*** At the present date, no; you could set the callback function to NULL, and monitor your own window messages. However, this could add needless OS-specificity to your client, and has other disadvantages that are discussed in question 4.1.2.

### 4.1.5 When I have an overlay resource, and am using to to display my rendered buffers, can I insert the buffer swap packets into the command fifo?

***Ryan:*** No. The problem is that you might lose ownership of the overlay resource before you get a chance to insert those packets into the actual command stream. You have to do this step after calling *csExecuteCommands()*, and in lieu of *csSwapBufferToDisplay()*. (rvb)

## 4.2 Glossary

### 4.2.1 Exclusive Mode

When a client is running in Exclusive Mode, it has full ownership of the underlying rendering and rasterization hardware. A client is said to be in Exclusive Mode if it meets all of the following requirements:

- ***The window it has declared as its rendering target currently has focus***
- ***The client area of the window that the CS client has declared as its rendering target originates at position (0,0) on the desktop, and its size corresponds exactly to the size of the desktop.***

Since these parameters can change independently and asynchronously of the client, Central Services must inform each client when they gain or lose exclusivity. This is done via the state callback function that was associated with the client's window at csGetGraphicalContext() time.

Here is a list of some of the things a client can lose ownership of, when it loses exclusivity:

- ***Overlay resources***
- ***Render buffers***
- ***Texture buffers***

### 4.2.2 Primary Surface

The region in the frame buffer that the backend graphics hardware uses as the primary input to the rasterizer. (It may mix in various off-screen overlay surfaces as well, but these are not considered the primary surface… there can be only one.)

### 4.2.3 Command Buffer

Central Services clients use command buffers to store commands to be executed. They can be submitted for execution by the XQTBUFFER or RSTRXQTBUFFER protocol requests.

### 4.2.4 Sentinel Buffer

Sentinel buffers are used to indicate that a particular event in an asynchronous command stream has occurred. The most common use is to end a command buffer with an LFB command to write a serial number to a place in memory (the sentinel buffer). The number to be written is a serial number for the piece of the command buffer that stores the LFB command. Using this, the client can determine if the subsection of the buffer has executed and is therefore free for reuse.

### 4.2.5 State Restoration Buffer

State restoration buffers restore the state expected at the *beginning* of a command buffer. The command buffer may then modify the state from that point.

## 4.3 Client-Managed Texture Heaps

TBD.

## 4.4 Debugging

Debugging messages are always available no matter how the library is built. To enable debug output use the following environment variables:

**CSDEBUG_FILE=DEBUG** to output to a debugger, ***filename*** to output to a file
**CSDEBUG_LEVEL=**Maximum level of debug output to see. Various levels of debug are avaialable.