Abstract Platform Layer

Targeting new V3D platforms

Broadcom Corporation
2nd February 2012

CONFIDENTIAL

Version 0.3

Contents

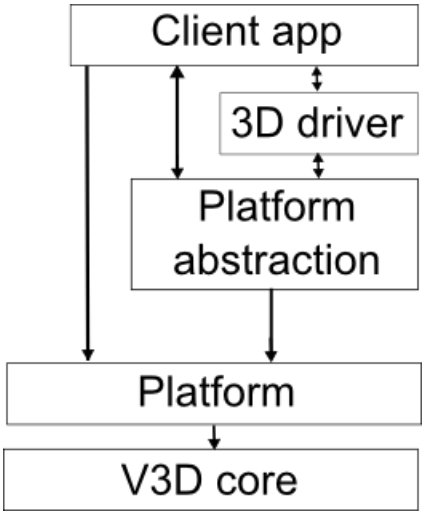
Document History

Version	Comments	Date	Editor
0.1	First Draft	1st Feb 2011	Gary Sweet
0.2	Added new API functions	31st Aug 2011	Gary Sweet
0.3	Documented changes for multiple windows & threads	2nd Feb 2012	Gary Sweet

1 Introduction

Broadcom's V3D graphics core is available in many different SoC solutions for several different markets. As such, it will need to interface to several different underlying platforms.

In order to avoid having to support multiple platforms in the EGL driver code directly, we have implemented an abstract platform support layer. In order to support a new platform, a new platform layer implementation is needed, but driver changes are not required. This document provides a high-level overview of the abstraction.



Since the platform layer is linked as part of the application, not the driver, custom versions can be created where specialized behavior is required (dual display support for example). It is sensible to provide a default implementation of the interface for a platform that most applications can use, but the implementation can be custom where required.

The platform layer encapsulates three specific areas of functionality:

1. Device memory allocation
2. V3D hardware interfacing (essentially register reads & writes)
3. Display buffer management

2 Registering the platform

In order to use the V3D EGL/GLES driver the application must register its platform interface with the driver. This is done via an extra API exposed from the driver library.

```
void BEGL_RegisterDriverInterfaces(BEGL_DriverInterfaces *iface);
```

The driver interface structure points at the three specific interfaces that are being hooked out and provides some callback pointers for the display layer.

```
typedef struct
{
   BEGL_MemoryInterface  *memInterface;
   BEGL_HWInterface      *hwInterface;
   BEGL_DisplayInterface *displayInterface;

   BEGL_DisplayCallbacks displayCallbacks;/* For client to callback into driver */

} BEGL_DriverInterfaces;
```

The individual interfaces will be described below.

3 Device Memory Allocation

The V3D driver needs to allocate and manage blocks of device memory for everything that is accessed by the V3D core. This includes render buffers, textures and control lists for example. All device memory allocation in the driver is passed through this abstract interface.

```
typedef struct
{
   /* Context pointer - opaque to the 3d driver code. Passed out in all calls. */
   void *context;

   /* Allocate aligned device memory, and return the cached address */
   void *(*Alloc)(void *context, size_t numBytes, uint32_t alignment);

   /* Free a previously allocated block of device memory. Pass a cached address.*/
   void (*Free)(void *context, void *pCached);

   /* Return a physical device memory offset given a cached pointer. */
   uint32_t (*ConvertCachedToPhysical)(void *context, void *pCached);

   /* Return a cached memory pointer given a physical device memory offset. */
   void *(*ConvertPhysicalToCached)(void *context, uint32_t offset);

   /* Converts a cached to an uncached pointer */
   void *(*ConvertCachedToUncached)(void *context, void *pCached);

   /* Converts an uncached pointer to a cached pointer */
   void *(*ConvertUncachedToCached)(void *context, void *pUncached);

   /* Flush the cache for the given address range.*/
   void (*FlushCache)(void *context, void *pCached, size_t numBytes);

   /* Retrieve information about the memory system
      Type can be one of :
       BEGL_MemHeapStartPhys, BEGL_MemHeapEndPhys, BEGL_MemCacheLineSize,
       BEGL_MemLargestBlock, BEGL_MemFree
   */
   uint32_t (*GetInfo)(void *context, BEGL_MemInfoType type);

   /* Copy data as fast as possible. Might use DMA where available (if both
      pointers are in contiguous device memory) */
   void (*MemCopy)(void *context, void *pCachedDest, const void *pCachedSrc,
                   uint32_t bytes);

} BEGL_MemoryInterface;
```

The platform layer creator (or application developer) is free to use whatever mechanism is appropriate on their platform to implement these functions. Nexus platforms, for example, use the Nexus heap memory manager to fulfill these requests.

4 Hardware Interfacing

```
typedef struct
{
   /* Context pointer - opaque to the 3d driver code. Passed out in all calls. */
   void *context;

   /* Acquire exclusive access to the 3D hardware */
   bool (*Acquire)(void *context, const BEGL_HWAcquireSettings *settings);

   /* Release access to the 3D hardware */
   bool (*Release)(void *context);

   /* Issue command to the 3D hardware. You must acquire first. */
   /* Will map to a register write somewhere in the implementation. */
   void (*SendCommand)(void *context, BEGL_HWCommand command, uint32_t data);

   /* Read status from 3D hardware. */
   /* Will map to a register read somewhere in the implementation. */
   uint32_t (*ReadStatus)(void *context, BEGL_HWCommand status);

} BEGL_HWInterface;
```

The driver will typically ask to `Acquire` the hardware via this interface for exclusive use while it completes one task item (usually a frame of rendering). Once the frame is complete, it will `Release` the hardware again, at which point it might be acquired by another client.

`SendCommand` and `ReadStatus` calls are only allowed whilst the hardware is acquired.

5 Display Buffer Management

This is the most complex of the platform interfaces. It manages the creation and destruction of render buffers, along with the presentation of rendered buffers when they become available.

The V3D hardware has certain constraints on its render buffers in terms of memory alignment and padding. Because of this, the driver must either be responsible for creating the render buffers, or the driver must be queried for constraints that the platform layer or application must respect when making buffers. The driver doesn't care about how the buffer memory is wrapped – so you can create a native surface type when asked for a buffer, provided that the alignment and padding constraints are met for the underlying memory.

One big advantage of exposing the display management via the abstract interface is that is allows you to take advantage of the performance of triple buffering even if you ultimately want to composite the render buffer onto the display. Normally, you would use pixmap rendering to get a native format render buffer for compositing. That mechanism will still work using the abstract API, but is always inefficient, since you have to flush and wait for the gl pipeline using glFinish() before you can composite the buffer.

Because the buffers for the internal or external multi-buffered swap chain are managed through this abstraction, you can simply continue to use eglSwapBuffers() to signal the end of a frame. The BufferDisplay function will be called when a buffer is ready to present, which can then be presented as required.

The display interface is as follows, the functions will be described in detail below, so are not commented here for brevity.

```
typedef struct
{
   /* Context pointer - opaque to the 3d driver code. Passed out in all calls. */
   void *context;

   BEGL_BufferHandle (*BufferCreate)(void *context,
                                     BEGL_BufferSettings *settings);

   BEGL_BufferHandle (*BufferGet)(void *context,
                                  BEGL_BufferSettings *settings);

   BEGL_Error (*BufferDestroy)(void *context,
                               BEGL_BufferDisplayState *bufferState);

   BEGL_Error (*BufferAccess)(void *context,
                              BEGL_BufferAccessState *bufferAccess);

   BEGL_Error (*BufferDisplay)(void *context, BEGL_BufferDisplayState *state);

   BEGL_Error (*BufferGetCreateSettings)(void *context, BEGL_BufferHandle buffer,
                                         BEGL_BufferSettings *settings);

   void *(*WindowPlatformStateCreate)(void *context, BEGL_WindowHandle window);

   BEGL_Error (*WindowPlatformStateDestroy)(void *platformState);

   BEGL_Error (*WindowUndisplay)(void *context, BEGL_WindowState *windowState);

   BEGL_Error (*WindowGetInfo)(void *context, BEGL_WindowHandle window,
                               BEGL_WindowInfo *info);

   BEGL_Error (*IsSupported)(void *context, BEGL_BufferFormat bufferFormat);

   BEGL_Error (*GetNativeFormat)(void *context, BEGL_BufferFormat bufferFormat,
                                 unsigned int * nativeFormat);

   BEGL_Error (*SetDisplayID)(void *context, unsigned int displayID,
                              unsigned int *outEglDisplayID);


} BEGL_DisplayInterface;
```

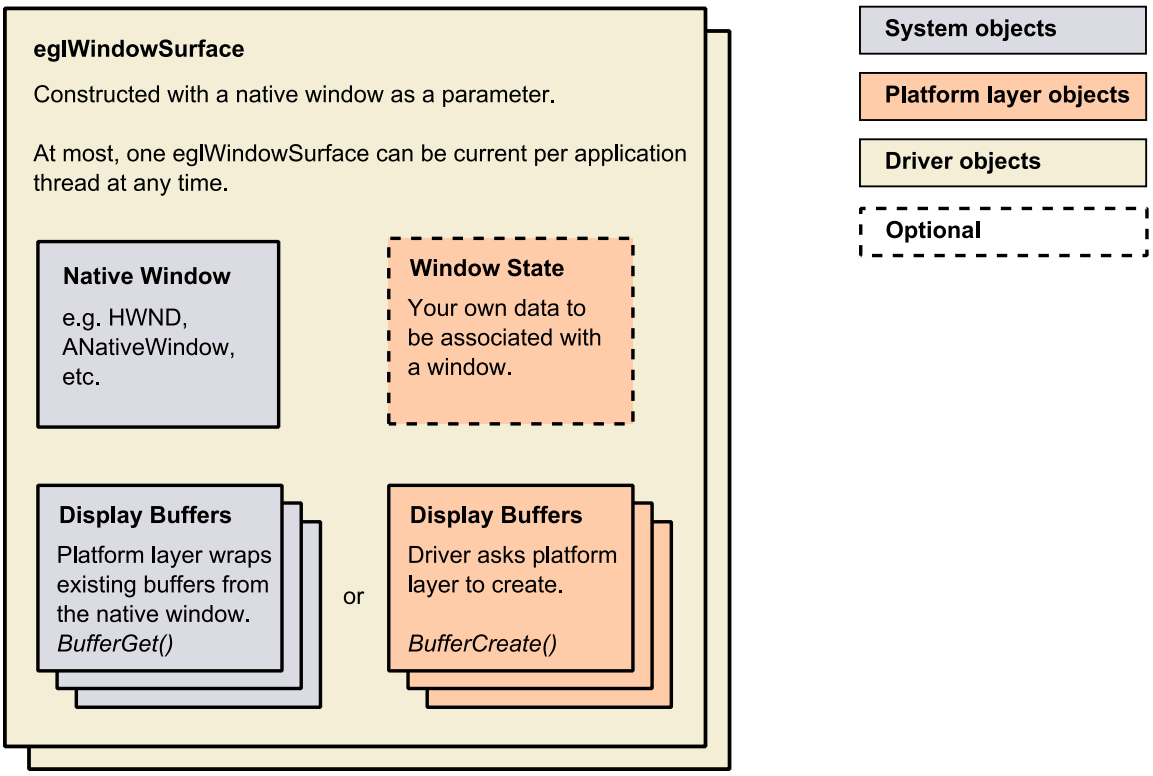
The abstraction layer has to be able to support many varied underlying platforms, each of which might have different capabilities and requirements. Current implementations include Nexus exclusive display, Nexus surface compositor display, DirectFB, Android App, Android Compositor, Win32 & Carbon.

5.1 Swap-chain creation

Early versions of the platform layer interface only supported swap-chain buffers being created by the V3D driver. Current versions allow wrapping of buffers created outside of the driver for use in the swap chain.

The latest version of the display interface now also supports multi-threaded applications with multiple window surfaces.

The EGL driver needs to obtain a number of display buffers to use as a swap-chain for its rendering. This process is triggered when eglCreateWindowSurface is called by the application.



Each EGL window surface is connected to a single native window. Attempting to connect multiple EGL window surfaces to the same native window will result in an error.

Rarely, applications will render into multiple native windows. This typically implies that a surface compositing platform is in play. Your platform layer should account for this if you want to support multi-windowed applications. You may have multiple eglWindowSurfaces, each with its own set of swap-chain buffers. Multiple windows can also be written simultaneously from multiple threads.

An application may also do multi-threaded rendering, where all but one thread is rendering off-screen, into FBOs or Pbuffers.

Your platform layer can be written to either provide the swap-chain buffers from pre-allocated surfaces (providing the alignment and size constraints are met), or to create the buffers on demand. These two options are described in more detail below.

In addition to the swap-chain buffers, you might find it useful to be able to hold your own arbitrary data associated with the window/swap-chain. This is labeled Window State in the diagram above. This can be achieved by implementing the `WindowPlatformStateCreate()` and `WindowPlatformStateDestroy()` functions. See the function descriptions below for more detail.

5.1.1 Driver created swap-chain

To have the driver create the swap-chain buffers automatically, you should set the `swapchain_count` member of the `BEGL_WindowInfo` structure to 0 (zero) when `WindowGetInfo()` is called.

`BufferCreate()` will be called by the driver multiple times to create the buffers used for the swap chain, along with any FBOs or pixmaps.

If you are setting `swapchain_count` to zero and letting the driver create the swap chain you do not need to provide an implementation for `DispBufferGet()` or `BufferAccess()` – simply leave the function pointers NULL.

5.1.2 External swap-chain

To use buffers that have already been created outside of the V3D driver, you will set the `swapchain_count` member of the `BEGL_WindowInfo` structure to the number of buffers in the swap-chain (usually 2 or 3) when `WindowGetInfo()` is called.

Note: The externally created buffers must still respect the size and alignment constraints required by the V3D driver. These constraints should be queried from the driver using the `BufferGetRequirements()` function pointer of the display callbacks. The requirements may change from release to release, so should always be queried dynamically.

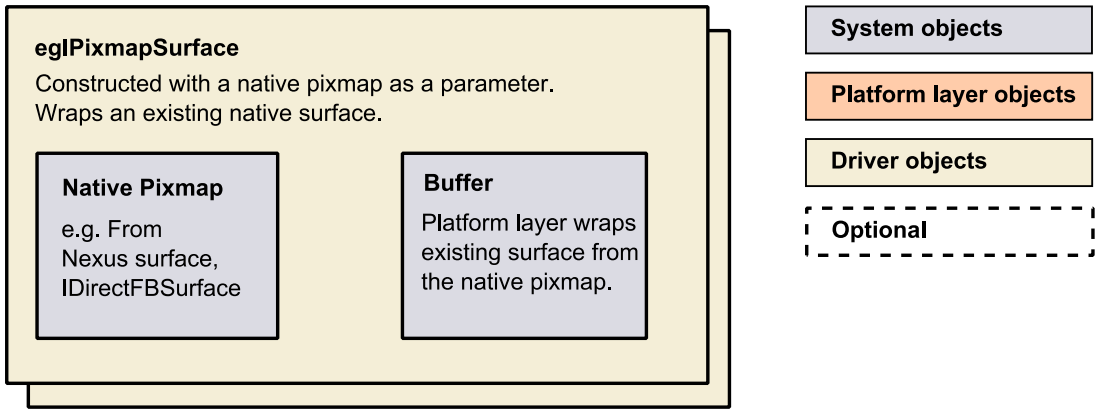
`DispBufferGet()` will be called for each swap-chain buffer. You should return the appropriate externally created buffer as requested.

When using an external swap-chain, the driver will make calls to your `BufferAccess()` function in order to lock buffers for rendering and unlock them when done. When the driver requests that a buffer is locked, you must return the physical device address and cached address of that buffer. You should not trigger a buffer display on an unlock, the `BufferDisplay()` function will be called at the appropriate time by the driver.

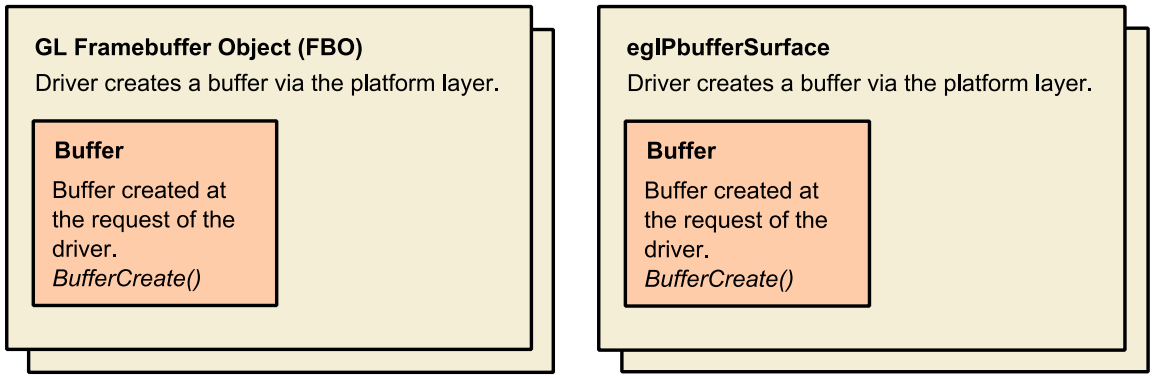
`DispBufferCreate()` will not be called by the driver to create the buffers used for the swap chain, however, it **will** still be called to create other buffers such as FBOs and pixmaps, so you will still need an implementation.

5.1.3 Other render targets

Pixmap surfaces wrap existing native surface formats and can be rendered in a single-buffered fashion. Before using the surface in a native API you must call `glFinish()` to ensure that the rendering is complete. Since this call blocks until done, it is not very efficient; the 3D core cannot render ahead. For this reason, we discourage the use of pixmap rendering.



FBO and Pbuffers are surfaces that are created and used only by GL. The buffers used for this rendering are still created via the `BufferCreate()` function.



5.2 Function Descriptions

BEGL_Error BufferDisplay(void *context, BEGL_BufferDisplayState *state)

Required

Request that the given state->buffer be displayed or added to a display queue.
When the buffer has actually been displayed (which implies that the previous one is free to be reused) you must call BEGL_Buffer_OnDisplay() with the same state->cb_arg. You might call BEGL_Buffer_OnDisplay() on the vsync signal following a SetFramebuffer call for example.

Notes:

1. You may get multiple calls to BufferDisplay() with the same buffer. You should call BEGL_Buffer_OnDisplay() each time (again, during the vsync normally). This is essentially a 'keep on display/still on display' sequence.
2. Normally, you will only receive a new BufferDisplay() after you have called BEGL_Buffer_OnDisplay() for the previous frame. However, if the user sets SwapInterval(0) in EGL, BufferDisplay() will be called as quickly as frames are available. Just call BEGL_Buffer_OnDisplay() at vsync time as usual in this case. This mode of operation is not recommended (it's a performance measuring tool). It should not be used at all when compositing, only when setting the framebuffer.

You do not need to, and for performance reasons should not block waiting for a vsync before returning from this call.

BEGL_BufferHandle BufferCreate(void *context, BEGL_BufferSettings *settings)

Required

Request creation of an appropriate displayable buffer, which must be created using the constraints in the settings parameter.

We could have just requested a block of memory using the memory interface, but by having the platform layer create a 'buffer' it can actually create whatever type it desires directly, and then only have to deal with that type. For example, in a Nexus platform layer, this function might be implemented to create a NEXUS_Surface (with the correct memory constraints of course).

When the buffer handle is passed out during BufferDisplay, the platform layer can simply use it as a NEXUS_Surface. It doesn't have to wrap the memory each time, or perform any lookups. Since the buffer handle is opaque to the 3d driver, the
platform layer has complete freedom. You might, for example, wrap the native surface in your own structure in order to store extra data alongside the native buffer.

This function will be used to create buffers for pixmaps, framebuffer objects and pbuffers. If you are not using an externally created swap-chain, it will also be used to create the swap-chain buffers for each window surface.

```
BEGL_BufferHandle BufferGet(void *context, BEGL_BufferSettings *settings)
```

Optional

Request a buffer from a pre-allocated swap chain. The length of the chain is returned via WindowGetInfo() and the external swap chain must contain the correct number of buffers to match the swap chain length.

You may pass 0 as the swap-chain length to have the driver create the buffers. In this case, you do not need to provide an implementation for this function.

```
BEGL_Error BufferDestroy(void *context, BEGL_BufferDisplayState *bufferState)
```

Required

Destroy a buffer previously created with BufferCreate().

The buffer will not be used by the driver after this call has been made, so you can safely delete the resource.

```
BEGL_Error BufferGetCreateSettings(void *context, BEGL_BufferHandle buffer,
                                   BEGL_BufferSettings *outSettings)
```

Required

Get information such as width, height, memory pointers etc. about a created window buffer.

```
BEGL_Error BufferAccess(void *context, BEGL_BufferAccessState *bufferAccess)
```

Optional

Called to lock or unlock a buffer in an external swap chain.
The driver locks the buffer before rendering into it. The lock must return the memory pointers for the buffer.

```
BEGL_Error IsSupported(void *context, BEGL_BufferFormat bufferFormat)
```

Optional

Check if the platform can support the buffer format passed in.

A little-endian platform is required to support `BEGL_BufferFormat_eA8B8G8R8`, `BEGL_BufferFormat_eX8B8G8R8` and `BEGL_BufferFormat_eR5G6B5`, so these are not checked.

A big-endian platform is required to support `BEGL_BufferFormat_eR8G8B8A8`, `BEGL_BufferFormat_eR8G8B8X8` and `BEGL_BufferFormat_eR5G6B5`.

If your platform only supports these formats, you do not need to provide this function.

```
BEGL_Error GetNativeFormat(void *context, BEGL_BufferFormat bufferFormat,
                           unsigned int *outNativeFormat)
```

Optional

Provide support for the EGL API eglGetConfigAttrib(), which translates an EGL config ID into a platform specific render type. The typical use-case would be to select a config ID and create a matching native window of that type, providing the native window to eglCreateWindowSurface().

```
BEGL_Error SetDisplayID(void *context, unsigned int displayID,
                        unsigned int *outEglDisplayID)
```

Optional

Called when eglGetDisplay() is called. Used in cases such as X11, so the X display can be passed into the driver. Could also be used in NEXUS to remove the need for an app to explicitly call it's platform registration.

```
BEGL_Error WindowGetInfo(void *context, BEGL_WindowHandle window,
                         BEGL_WindowInfoFlags flags, BEGL_WindowInfo *outInfo)
```

Required

Called to determine current size of the window referenced by the opaque window handle. This is needed by EGL in order to know the size of a native 'window'.

```
BEGL_Error WindowUndisplay(void *context, BEGL_WindowState *windowState)
```

Required

Flush all pending display of all of the swap-chain buffers in this window until they are all done, then remove the window from display. Should block until complete. Buffers and windows might be destroyed directly following this call, so ensure buffers are no longer in use when you return.

```
void *WindowPlatformStateCreate(void *context, BEGL_WindowHandle window)
```

Optional

Called prior to any swap-chain buffers being created for the given window.

If you need to store any state per window (swap-chain) then you can allocate a structure of your creation in this function and return it.

The `BEGL_WindowState->platformState` that is available in the arguments to most of the platform layer functions will contain the pointer that you return here for the associated window.

`BEGL_Error WindowPlatformStateDestroy(void *platformState)`
Optional (required if WindowPlatformStateCreate() is implemented) Called when your custom window state structure needs to be destroyed. The structure will not be used after this.

6 Summary

The abstract platform layer is used by the graphics driver for all platform level operations to avoid the need for the driver itself to support new platforms.

There are a number of implementations already which can be found in vobs\rockford\middleware\platform.

The Nexus, Android and DirectFB implementations should be useful references.

7 Changes from previous version

7.1 Device memory interface

New parameters accepted by `GetInfo()`:

- `BEGL_MemLargestBlock – should return the largest free memory block size`
- `BEGL_MemFree – should return the total amount of free memory available`

New **optional** interface function:

- `MemCopy – can be used to drive a fast DMA based device memory copy if one is available (and actually faster than a CPU based copy!).`

7.2 Display interface

The display interface was cleaned up a little and extended to support multiple windows and multi-threading. A new window state structure is available in the arguments of many of the interface functions, or via their existing structure arguments, to enable consistent access to the window handle and custom window state structure.

```
typedef struct
{
   void                *platformState; /* Custom window state pointer */
   BEGL_WindowHandle   window;

} BEGL_WindowState;
```

The following display interface functions have been **deprecated** and will no longer be called by the latest driver code:

- `BufferDisplayFinish – replaced by WindowUndisplay`
- `BufferGetWin – no longer required`

New **required** interface functions:

- `WindowUndisplay`

New **optional** interface functions:

- `WindowPlatformStateCreate`
- `WindowPlatformStateDestroy`
- `SetDisplayID`

Changed arguments:

OLD	`BEGL_BufferHandle (*BufferGet)(void *context, BEGL_WindowHandle window, BEGL_BufferSettings *settings);`
NEW	`BEGL_BufferHandle (*BufferGet)(void *context, BEGL_BufferSettings *settings);`
Changed to match `BufferCreate()` arguments. Window can now be obtained from `settings->windowState.window`	

OLD	`BEGL_Error (*BufferDestroy)(void *context,` `  BEGL_BufferHandle buffer);`
NEW	`BEGL_Error (*BufferDestroy)(void *context,` `  BEGL_BufferDisplayState *bufferState);`
Changed to match `BufferDisplay()` arguments. Buffer handle can now be obtained from `bufferState->buffer`	