Nexus
Graphics

Broadcom
Corporate Headquarters: San Jose, CA

Web: www.broadcom.com

Revision History

Revision	Date	Change Description
0.1	3/16/11	Initial draft
0.2	2/5/13	Update
0.3	6/7/13	Minor update
1.0	2/14/17	Added graphics compression
1.1	5/11/18	Added Reducing Graphics Power

Table of Contents

Introduction

This document describes techniques needed to implement a 2D graphics system using Nexus. It is assumed that the reader has general knowledge of nexus, and run and studied the nexus/examples/graphics applications, and therefore can build and run a simple nexus graphics application.

The intended audience is the nexus user. Nexus has the following graphics interfaces:

Capability	Explanation
NEXUS_SurfaceHandle	Handle to a graphics surface
NEXUS_Graphics2DHandle	Handle to the 2D graphics blitter (M2MC HW)
NEXUS_DisplayHandle	Handle to a display which can have a graphics feeder (GFD HW). Any surface set to a display is called a *framebuffer*.
NEXUS_Graphics3DHandle	3D interface. Note that Nexus does not have a 3D API. Instead, Broadcom uses OpenGL-ES 2.0 as its 3D API.

In order to explain how certain nexus interfaces work, there may be some reference to lower-level Magnum API's or actual hardware blocks. The following is a mapping:

Nexus Interface	Magnum Module	Hardware Block
Surface	PXL	DRAM
Graphics2D	GRC (Graphics compositor)	M2MC (Mem-to-mem compositor)
Display	VDC (Video Display Controller)	GFD (Graphics feeder)
Graphics3D	V3D, VC5	VC4 and VC5 (VideoCore IV and V)

This document does not cover the following graphics topics:

Topic	Details
NEXUS_PictureDecoder	Decode JPEG, PNG and GIF images using HW. See Nexus_Usage.pdf for documentation.
NEXUS_StillDecoder	Decode MPEG, AVC and HEVC stills into a surface using video decoder HW. See Nexus_Usage.pdf for documentation.
3D Graphics	This is supported through OpenGL-ES 2.0. See nexus/docs/OpenGL_ES_QuickStart.pdf.
External graphics libraries	Broadcom provides a separate AppLibs release with sample integration of libraries like DirectFB.
Fonts	Nexus has no direct support for fonts. See the AppLibs release for libraries like DirectFB.

Blitting

The basic 2D graphics operation is blit (block transfer) and fill. The NEXUS_Graphics2D interface supports a variety of blit and fill operations including scaling, alpha-blending, porter-duff operations, pixel conversion and color key. Please see API comments in nexus_graphics2d.h and example code in nexus/examples/graphics for details.

Opening multiple virtual blitters

On most chips, there is only one physical blitter (i.e. one M2MC HW block). However, NEXUS_Graphics2D provides a virtual interface to that blitter. Multiple threads or processes can open their own instance of NEXUS_Graphics2D and submit blits independent of each other. Nexus will serialize the calls and provide per-context Checkpoints as needed.

See nexus/examples/graphics/multithreaded_fill.c.

Synchronizing blits with Checkpoint

The 2D Graphics Interface provides blit (block transfer) and fill operations. These operations can be performed on any Nexus Surface, whether it is a frame buffer or an off-screen buffer.

Blits are asynchronous by default. In order to synchronize, you can request a “checkpoint” callback to know when all previously issued blits have completed. This checkpoint allows an application to interleave blits and any CPU-based drawing to the surface without overwriting each other. Refer to nexus/examples/graphics/blit.c for a working example.

A checkpoint callback works as shown in Checkpoint Callback.

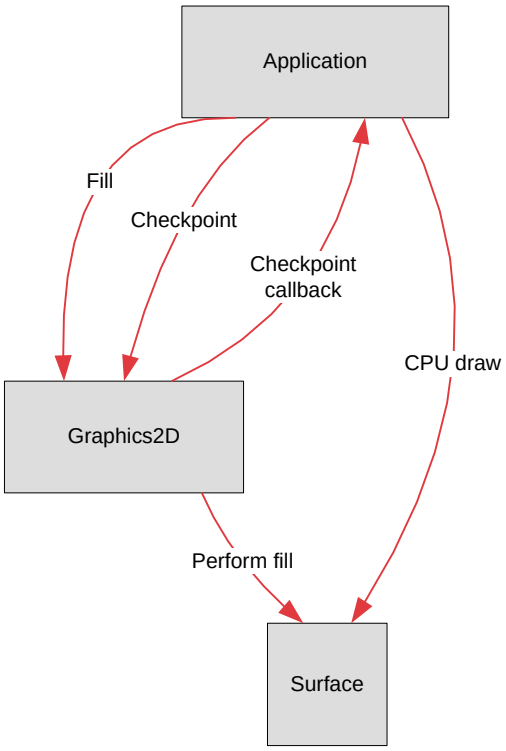


Figure 38: Checkpoint Callback

Another purpose of checkpoint is to flush the software fifo. By default, NEXUS_Graphics2DOpenSettings.packetFifoThreshold is non-zero, which means each Blit or Fill call will not be submitted immediately to the underlying M2MC hardware. It will be queued until either the threshold is met, the fifo wraps, or a checkpoint is called. So, making one call to blit with no checkpoint will result in no blit being performed. Nexus detects this condition and issues a “missing checkpoint” warning after 5 seconds. The proper fix is to add the checkpoint.

Optimal use of Checkpoint and Flush

All Nexus blits are asynchronous[1]. An application can synchronize the blit engine by calling NEXUS_Graphics2D_Checkpoint. For optimum performance, the application should only execute a Checkpoint when essential. Extra checkpoints decrease system performance because it causes the blit pipeline to necessarily run dry which results in the M2MC HW being idle. Checkpoints are only required right before a CPU direct access or right before setting a surface as the framebuffer.

Likewise, all direct CPU access to surface memory is through cached memory. An application can flush cached memory by calling NEXUS_Surface_Flush. For optimum performance, the application should only execute a flush when essential. Extra flushes decrease system performance because it is a synchronous operation that requires many CPU clock cycles. Flush is only required before a HW access, which is either blitting to a surface, blitting from a surface, or setting a surface as the framebuffer.

The key point is that an app cannot know if a checkpoint is needed **after** a blit, or flush is needed **after** a cpu access, until it sees the next operation. Unless it knows the next operation, the checkpoint or flush may be unnecessary and therefore inefficient. You can only know if a checkpoint is required **before** doing a cpu access which immediately follows a blit, or a flush is required **before** doing a blit which immediately follows a CPU access.

The following pseudo-code illustrates this:

```
Blit(surface)
Blit(surface)
Blit(surface)
Blit(surface)

/* call Checkpoint before mem.buffer access */
Checkpoint
GetMemory(surface, &mem)
mem.buffer[x+mem.pitch*y+0] = 1;
mem.buffer[x+mem.pitch*y+1] = 1;
mem.buffer[x+mem.pitch*y+2] = 1;
mem.buffer[x+mem.pitch*y+3] = 1;

/* call Flush before Blit */
Flush(surface)
Blit(surface)
Blit(surface)
Blit(surface)
Blit(surface)
```

[1] In previous versions of nexus, NEXUS_Graphics2DSettings.blockedSync provided synchronous blits, but that API was never recommended and has now been deprecated.

```
/* call Checkpoint before SetGraphicsFramebuffer */
Checkpoint
NEXUS_Display_SetGraphicsFramebuffer(surface)
```

Given this principle, the following is an inefficient routine:

```
void draw_button(surface) {
    Blit(surface)
    Blit(surface)
    Checkpoint /* this is required */
    GetMemory(surface, &mem)
    mem.buffer[x+mem.pitch*y+0] = 1;
    mem.buffer[x+mem.pitch*y+1] = 1;
    Flush(surface); /* this could be unnecessary */
}
```

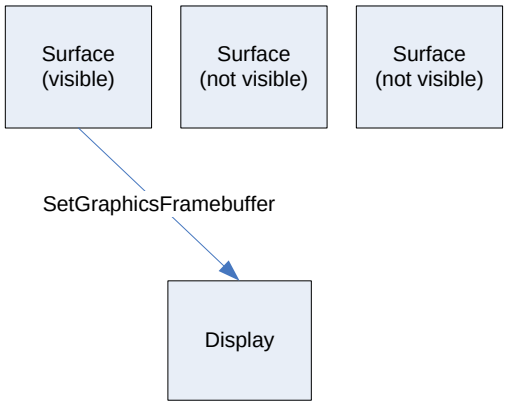
There should be some application state which allows the routine to be efficient:

```
void draw_button(surface) {
    if (app.last_op == eCpu) { /* read app state */
        Flush(surface); /* this is required */
    }
    Blit(surface)
    Blit(surface)
    Checkpoint /* this is required */
    GetMemory(surface, &mem)
    mem.buffer[x+mem.pitch*y+0] = 1;
    mem.buffer[x+mem.pitch*y+1] = 1;
    app.last_op = eCpu; /* save to app state */
}
```

Multi-buffering

Nexus does not implement multi-buffering; instead it is up to the application to manage framebuffer switches using a nexus function and callback. This puts more burden on the application, but also provides maximum flexibility and performance. However, it is important that the multi-buffering code be written correctly.

The application selects which NEXUS_Surface will be connected to the Graphics Feeder (GFD) HW and be displayed on the screen. This is done using NEXUS_Display_SetGraphicsFramebufer.



In most applications, a double and triple buffering scheme is needed to prevent rendering artifacts (tearing, tiling, etc.) being seen. This means that the application is responsible for managing a set of two, three or more surfaces which cycle as the framebuffer. The rules for proper framebuffer use are:

1. Before setting a new framebuffer, all updates to the framebuffer's memory must be complete. This requires flushing any CPU writes (NEXUS_Surface_Flush) and issuing a checkpoint for any pending blits (NEXUS_Graphics2D_Checkpoint)

2. After giving a framebuffer to the display, you cannot write to it again until you have given another framebuffer to the display and that this new framebuffer is being displayed. This is done by waiting for NEXUS_GraphicsSettings.frameBufferCallback[2].

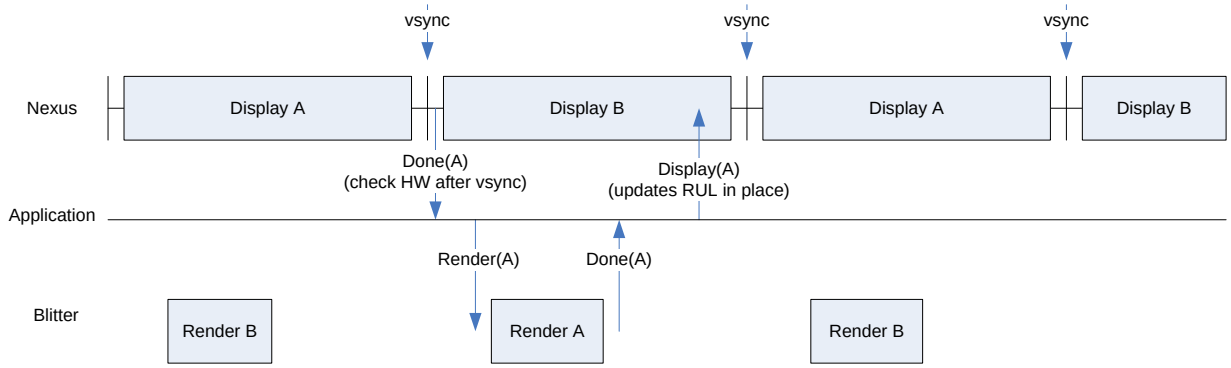
[2] You cannot use NEXUS_DisplaySettings.vsyncCallback because there is no guarantee that your new framebuffer will be applied on the next vsync. NEXUS_GraphicsSettings.frameBufferCallback actually verifies that the new framebuffer is actually applied using BVDC_Source_GetSurface_isr.

The number of framebuffers required depends on your desired frame rate and on your average render time. The following table gives typical options:

Render Time	Frame Rate	# of Buffers Required
Much shorter than 1 vsync (e.g. 2-5 milliseconds)	60 fps	2
1 full vsync	60fps	3
1 full vsync	30 fps	2

The following diagrams illustrate these options:

If you have a very short render time, you can double buffering and still achieve 60fps:



Some observations on the diagram:

- The Done() arrow from Nexus → Application corresponds to the Nexus framebufferCallback
- The Display() arrow from Application → Nexus corresponds to NEXUS_Display_SetGraphicsFramebuffer
- The Render() and Done() arrows between Application and Blitter correspond to any calls the app may do render into an non-displayed framebuffer.

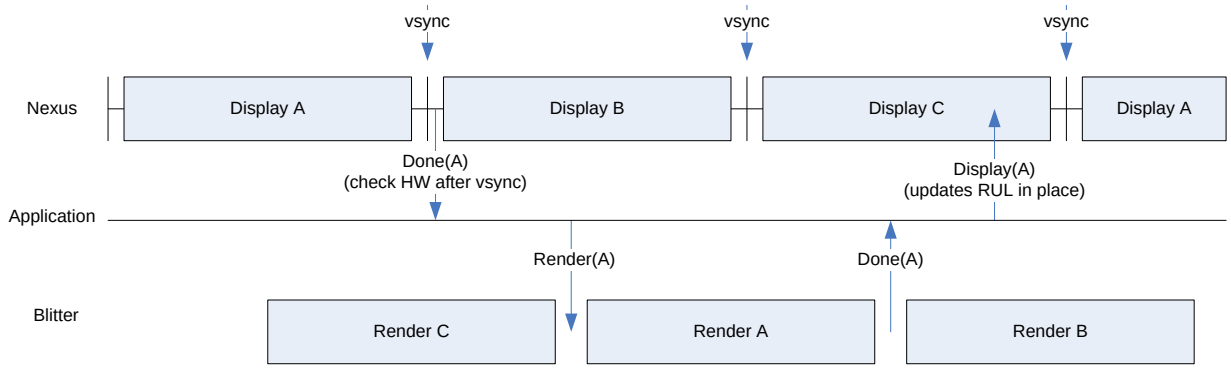
In the vsync ISR, nexus will check if the last framebuffer set has become visible. If true, it will fire the framebufferCallback. This implies that any previously set framebuffer is no longer visible.

When the app receives the framebufferCallback, it can start rendering into that previously set surface.

Nexus does not maintain a queue of framebuffers. Therefore the app must also wait for the framebufferCallback so that it doesn't overwrite a previously set framebuffer.

If the app can submit the next framebuffer (with NEXUS_Display_SetGraphicsFramebuffer) with enough time before the next vsync, Nexus is able to display that framebuffer on that vsync.[3] However, there is no guarantee that Nexus has set the framebuffer with "enough time". The only way for the app to know is to wait for the framebufferCallback.

If you require a full vsync's time to render, then 60 fps requires triple buffering:



However, if your rendering time exceeds 1 vsync, then no amount of buffering will allow you to achieve 60fps of updates. Triple buffering will give you the best possible performance, but it will be less than 60fps.

See nexus/examples/graphics/multibuffering.c for example code. You can set command line parameters of "-rendertime MSEC" and "-triple" options to see these modes in operation. The code also contains comments explaining how the API works.

[3] This is possible because BVDC_Source_SetSurface is able to update RUL's in place. Without this in place update, VDC would wait another vsync to display the new framebuffer.

Why manual double-buffering doesn't work

To avoid tearing, no update should be made to a visible framebuffer and all updates to the graphics feeder (including setting which surface is the current framebuffer) must be done in the vertical blanking interval (VBLANK). The VBLANK is a very short interval of time when no visible lines are being scanned, approximately 1 millisecond. However, applications using Nexus and Magnum do not have the ability to achieve this timing on their own.[4]

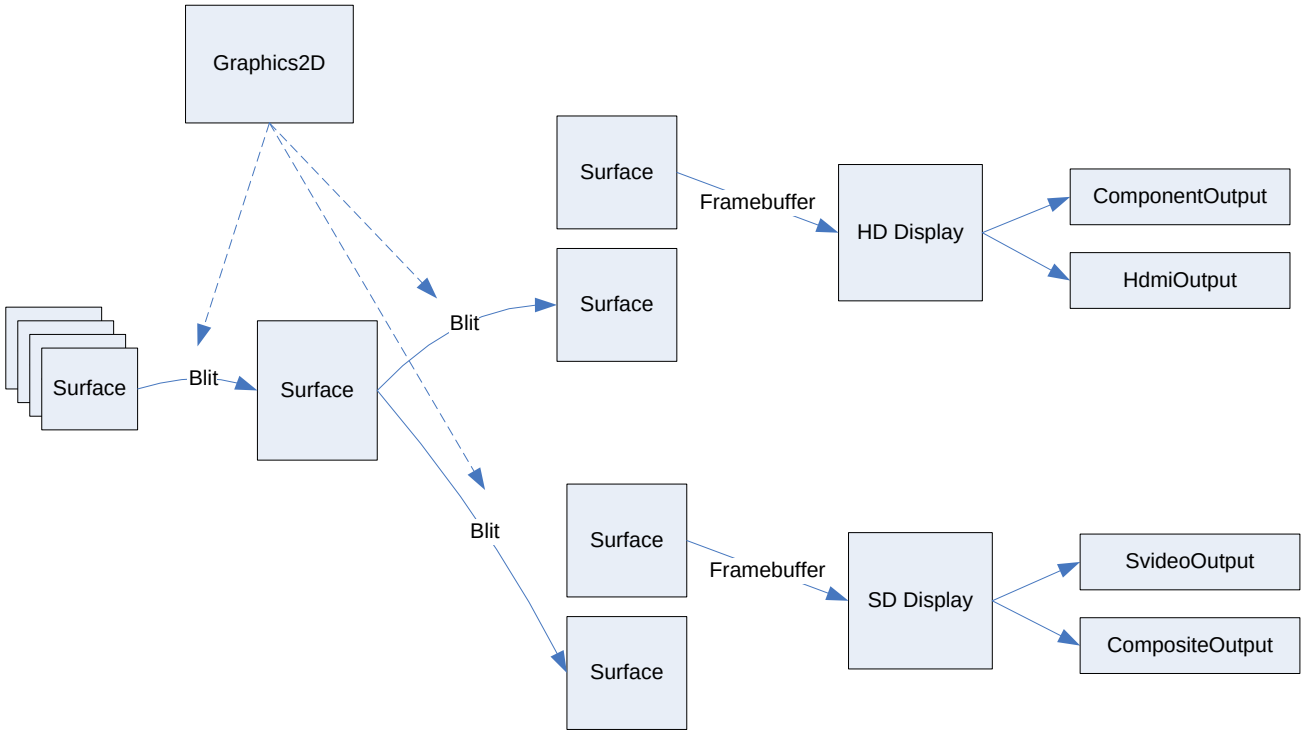
Instead, Nexus and Magnum use the RDC (Register DMA Controller) hardware to update the framebuffer during the VBLANK. The RDC is able to run a register update list (RUL) and program the graphics feeder (GFD) frame buffer address and other settings during the VBLANK. When an application calls NEXUS_Display_SetGraphicsFramebuffer, this RDC update is scheduled and the flip will happen during the next VBLANK. Using NEXUS_Display_SetGraphicsFramebuffer is the only way this capability is exposed.

If your application is not using NEXUS_Display_SetGraphicsFramebuffer, you will see tearing.

[4] This includes trying to do a direct CPU access to a visible framebuffer during the VBLANK. First, Nexus does not expose the ISR context to applications. While ISR context could conceivably be exposed in Linux user mode, it cannot be exposed in Linux kernel mode across the user/kernel boundary. Even if ISR context was exposed, the real-time scheduling tolerances of Magnum far exceed the approximate 1 millisecond VBLANK timing. Generally, the highest real-time requirement for Magnum is the 16 millisecond ISR for VDC, an order of magnitude greater than the VBLANK.

HD/SD Simul Systems

If your set-top supports simultaneous HD and SD displays, and if you want the same graphics content on both, you must write application code to set up both displays. A common technique is for the app to render into a single off-screen buffer, then blit twice, once into an HD framebuffer, a second time into an SD framebuffer. This allows you to handle double-buffering at the same time.



Blitter FIFO space

The blitter uses a FIFO of asynchronous blit requests. When this FIFO is filled, the blit call will fail. The application must trap this failure, wait for the packetSpaceAvailable callback, then re-issue the blit.

Many applications require a fixed amount of FIFO space for each context. They can simplify their code and optimize their performance by allocating a separate NEXUS_Grahics2DHandle for each context. Each handle provides a separate FIFO and a separate checkpoint callback. This provides for maximum isolation and performance.

Using macros for guaranteed FIFO space

If you still hit blit failures based on FIFO space, and if you have allocated separate blitters for each context, and if you have increased the FIFO space to the maximum level that you can, then these failures may be unavoidable. You will need to write a loop in your application as follows:

```
for(;;) {
  rc = fill();
  if(rc!= NEXUS_GRAPHICS2S_QUEUE_FULL) {
    break;
  }
  BKNI_WaitForEvent(packetSpaceAvailEvent, BKNI_INFINITE);
}
```

To ease the pain, we recommend using macros, with an app-specific blocking checkpoint, to make this transparent:

```
#define XXX_SAFE(op, packetSpaceAvailEvent, args) do { \
  NEXUS_Error rc; \
  rc = op(args); \
  if(rc!=NEXUS_GRAPHICS2S_QUEUE_FULL){break;}; \
  BKNI_WaitForEvent(packetSpaceAvailEvent, BKNI_INFINITE); \
  } while(1)

#define XXX_Fill_Safe(handle, settings) XXX_SAFE(NEXUS_Graphics2D_Fill,
packetSpaceAvailEvent, (handle, settings))

#define XXX_Blit_Safe(handle, settings) XXX_SAFE(NEXUS_Graphics2D_Blit,
packetSpaceAvailEvent, (handle, settings))
```

This technique must be done outside nexus because no NEXUS_Graphics2D call can be blocking.

Multi-plane Graphics

Many applications require multiple graphics planes. Broadcom hardware supports one graphics feeder (GFD) per compositor (CMP), however, a multi-plane graphics system can be implemented using the blitter and an intermediate buffer.

1. Blit a background image into an intermediate buffer.
2. Draw an alpha hole in the intermediate buffer where video will be.
3. Blit an OSD image with the intermediate buffer into the frame buffer.
4. Set the alpha for the OSD over the video hole in the frame buffer.

If the background and alpha hole does not change, then steps 1 and 2 do not need to be repeated for every OSD change.

Alpha-premultiplied surfaces can be used for more complex multiplane GUIs. Sample code is available from Broadcom upon request.

Graphics Compression

Some chips and box modes allow or require graphics compression to reduce GFD memory bandwidth. The M2MC will compress graphics if a blit's output surface is NEXUS_PixelFormat_eCompressed_A8_R8_G8_B8. A compressed surface cannot have any meaningful read or write access by the CPU.

See NEXUS_DisplayCapabilities.display[].graphics.compression to learn if compression is allowed or required.

If required, NxClient will configure Nexus SurfaceCompositor to automatically compress when composing the framebuffer.

If required, Nexus Display will also automatically compress a framebuffer if not already compressed. This will have a small effect on framebuffer update latency.

VC5 can also be configured to output compressed graphics.

Reducing Graphics Power

Graphics can consume significant power in a set-top box. The section gives pointers to Nexus and NxClient API's for reducing power and memory bandwidth for GFD and M2MC.

Disable GFD – When watching full screen video with no subtitles and closed captioning, there's no need for any graphics. Instead of having the GFD scan out 60fps of zeroes, we can disable GFD. For Nexus, call NEXUS_Display_SetGraphicsFramebuffer(display, NULL). For NxClient, there are several ways. Graphics can be disabled, regardless of input, by setting NxClient_DisplaySettings.graphicsSettings.enabled = false. Or, SurfaceCompositor will automatically disable graphics after a one second timeout if all clients have called NEXUS_SurfaceClient_Clear(). FUTURE: After one second of no updates, SurfaceCompositor will memory map and scan the pixels of an unsecure framebuffer and automatically disable if it is transparent.

Reduce frequency of GFD updates – M2MC power can be saved by writing the GUI to do fewer updates. Graphic designers may not realize that a small 60fps animation in the corner requires a 60fps full-screen re-composition of graphics.

Use BSTC compression – Both GFD and M2MC memory bandwidth is reduced by using BSTC compression. For some box modes, BSTC compression is required for the GFD and Nexus SurfaceCompositor will automatically use it for the framebuffer. However, BSTC compression can also be used by applications for surfaces where no CPU access is needed. See the "Graphics Compression" section earlier.

Use GFD upscale – By using less than full-resolution framebuffers, we reduce GFD and M2MC memory bandwidth and power. nxserver can be started with the "-fbsize W,H" parameter.

FUTURE: Use GFD positioning – When watching full screen video with subtitles or closed captioning, less than full-screen framebuffers may be sufficient, which results in GFD and M2MC savings. This requires the application to position SurfaceClients as less than full screen, or submit full screen surfaces with clipRects. SurfaceCompositor will take the union of all clipped surfaces and instruct GFD to only scan out that section.

Video as Graphics

Some applications need to process video data through the graphics pipeline. There are three methods for sampling a single picture from a stream (e.g. for a thumbnail). One of these methods is also suitable for rendering a video stream as a lower-resolution decode window (e.g. mosaics or PIP window on a single-window chip).

For each method, run and study the example application then read the API documentation in the appropriate header files.

Extraction from the display's multi-buffering pipeline

Call NEXUS_VideoWindow_CaptureVideoBuffer to get a picture from the display pipeline. This works for analog and digital sources. See nexus/examples/video/capture_buffer.c.

Capturing decoder output

Call NEXUS_VideoDecoder_CreateStripedSurface to get a surface for the most recently decoded picture. This requires destriping before use as raster graphics. For the video path, this is done by MFD. For the graphics path, this can be done by M2MC (blitter). See nexus/examples/video/decode_video_via_graphics.c. This is the only method that is suitable for rendering a video stream.

Still decode

Feed an encoded I-frame through the NEXUS_StillDecoder interface. See nexus/examples/graphics/still_decoder.c for a nexus example. Also see BSEAV/app/thumbnail for a sample extraction and rendering application.

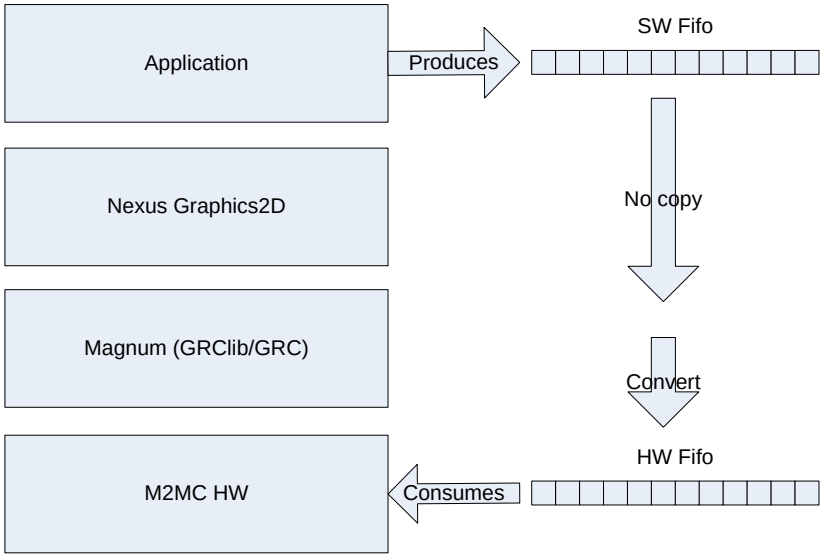
Packet Blit

Nexus has two API's for blitting. The simplest and safest API is the function-based API. This is primarily NEXUS_Graphics2D_Blit and NEXUS_Graphics2D_Fill. It is simple because the function parameters tell you what you must do. It is safe because the function will verify the parameters and serialize with any other blit.

However, the function-based API is inefficient if you are doing many small blits. The CPU overhead for setting up the blits may exceed the time required by the M2MC HW to blit, which means the M2MC HW will be under-utilized.

To address this, Nexus offers a data-driven API called packet blit. There are only two functions: NEXUS_Graphics2D_GetPacketBuffer and NEXUS_Graphics2D_PacketWriteComplete. The application's job is to get a buffer, fill it with data packets which tell the M2MC HW how to blit, then to submit that buffer. The format of the packets is specified by the bm2m_packet.h header file.

Packet blit has increased performance because it allows the application to build a pipeline of blits (called the SW FIFO) with a minimal amount of CPU set-up time per blit. This SW FIFO is also ideal for multi-process systems because the IPC traffic is minimized.



The packet blit API is neither simple nor safe, but it is fast. It is not simple because you must know what set of packets are required to accomplish a blit. If you don't specify all the packets, you will not get an error code. You just won't get a blit. You may corrupt system memory. It is not safe because there is little to no packet validation. The data in the packets is pushed into the HW and the blit is executed.

See nexus/examples/graphics/packet_blit.c and packet_fill.c for example code.

Packet Blit Safety and Debug

Because packet blit can be so dangerous, we've added some tools to provide gross-level safety and to debug.

You can ensure that packet blit only reads from or writes to a specific NEXUS_HeapHandle (which corresponds to a BMEM_Heap) by setting NEXUS_Graphics2DOpenSettings.boundsHeap. This is required for multi-process solutions and is set automatically for untrusted clients. This can also be helpful for application debug.

There are also two techniques that are not meant for safety, because they are inefficient, but can be used for system debug:

1)Compile with export BGRC_PACKET_VERIFY_SURFACE_RECTANGLE=y and use NEXUS_Surface_InitPlane. This will decrease performance, but will bounds check every blit's source, dest and output rectangles. You will see an error message in the console if something is out of bounds.

2)You can view the low-level register writes generated by the packet blit by hijacking bgrc_packet_priv.h and enabling #define BGRC_PACKET_P_DEBUG. A useful technique is to compare packet blit register writes with function-based blit register writes.

Improving Performance

There are a number of things that can be done in the application that will improve your graphics performance. Most involve caching information and minimizing the number of calls into Nexus. This is especially important in a multi-process environment where each function call has added overhead. Some of these ideas are repeated throughout this document but are summarized here.

- Call NEXUS_Surface_GetMemory after creating each surface and cache it with the surface handle. The information does not change.

- Cache the results of any GetDefaultSettings function that is called frequently. For instance, NEXUS_Surface_GetDefaultCreateSettings.

- Use separate NEXUS_Graphics2DHandle's in each graphics context. Don't allow one context to block waiting for another unrelated context. Allow each to run its own pipeline. This uses a small amount more of your system memory to improve performance. This is beneficial for packet blit and function-based blit.

- Use the packet blit API instead of NEXUS_Graphics2D_Blit(). If you are able to pipeline blits in your application, the performance improvement can be dramatic. See "Packet Blit" section of this document for more information.

- When using packet blit, don't call NEXUS_Graphics2D_SubmitPacket until either the FIFO is full or right before you are going to call NEXUS_Graphics2D_Checkpoint. Avoid calls to SubmitPacket which are premature and therefore unnecessary. If you run out of packet FIFO space because of the deferred SubmitPacket, increase the FIFO size as much as possible.

- Use a dedicated thread to call NEXUS_Display_SetGraphicsFramebuffer. Depending on other logic in the system, it could block in VDC waiting for a vsync. If your graphics thread needs to do other work (like CPU draws or more blits) it shouldn't wait on that.

- Consider creating off-screen surfaces to cache rendering work. This also uses more memory to improve performance.