



Nexus Graphics2D

Revision History

Revision	Date	Change Description
0.1	6/20/16	Initial draft

Table of Contents

Introduction	2
The Memory-to-Memory Compositor (M2MC).....	4
Common Operations.....	5
Fill	5
Copy	6
Blend	7
M2MC Overview	8
Internal Pixel Format.....	9
Source block.....	9
Destination block	10
Blend block.....	11
Pattern block.....	11
Output block	12
Blend Equations	12
Blend Factors	14
Example Blending Equations.....	14
Fill	14
Copy Source	15
Blend Source Over Destination	15
Visualizing Source Over Destination Blending	15
Common API	3
Procedural API.....	3
Packet Blit API	4
Packet Pipeline Visualization	4
Available Packet Types.....	6
Coding Style	7
Sample Call Sequence	8
Configuring Common Operations	10
Fill	10
Copy	11
Blend	14
Appendix 1: Pixel Formats	16
Appendix 2: Future Topics	18

Introduction

This document describes the features of Broadcom 2D graphics core, known as the memory-to-memory compositor (M2MC), describes how these features can be configured to perform specific 2D graphics composition operations, and describes how to implement these operations using the Nexus Packet Blit API.

It is assumed that the reader has general knowledge of nexus, and has 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 who needs to understand the 2D graphics core in order to implement graphics composition operations.

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 <i>framebuffer</i> .
NEXUS_Graphics3DHandle	3D interface. Note that Nexus does not have a 3D API. Instead, Broadcom uses OpenGL-ES 2.0 as its 3D API.

This document does not cover the following graphics topics:

Topic	Details
NEXUS_PictureDecoder	Decode JPEG, PNG and GIF images using Still Image Decoder (SID) HW. See Nexus_Usage.pdf for documentation.
NEXUS_StillDecoder	Decode MPEG, AVC and VC1 stills into a surface using video decoder (HVD) 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.

The Memory-to-Memory Compositor (M2MC)

The Broadcom 2D graphics core is known as the memory-to-memory compositor or M2MC. The M2MC is a general purpose pixel compositor. It is commonly referred to as a “blitter”. It can optionally read from one or two inputs and will always write to one output. The operation it performs on the pixels is highly configurable. Before going into depth on the M2MC, it helps to review the most common operations: fill, copy, and blend.

The M2MC accepts zero, one or two inputs and writes to one output. The pixels from the inputs can be transformed and combined in many different ways to generate many different operations.

When no inputs are used, the M2MC uses a constant color and this becomes a fill operation.

When one input is used, the operation can be as simple as a non-scaled copy. It can also be a scaled copy, a color space conversion, a format conversion, a colorkey operation, a filter operation, a shade operation, or any combination of these.

When two inputs are used, the operation will be some type of blend and/or colorkey, where the pixels from the two surfaces are combined. The specific combination is specified by the blending equation and the colorkey configuration. Nearly all UI implementations use some type of blending to composite the final UI from the UI components or images.

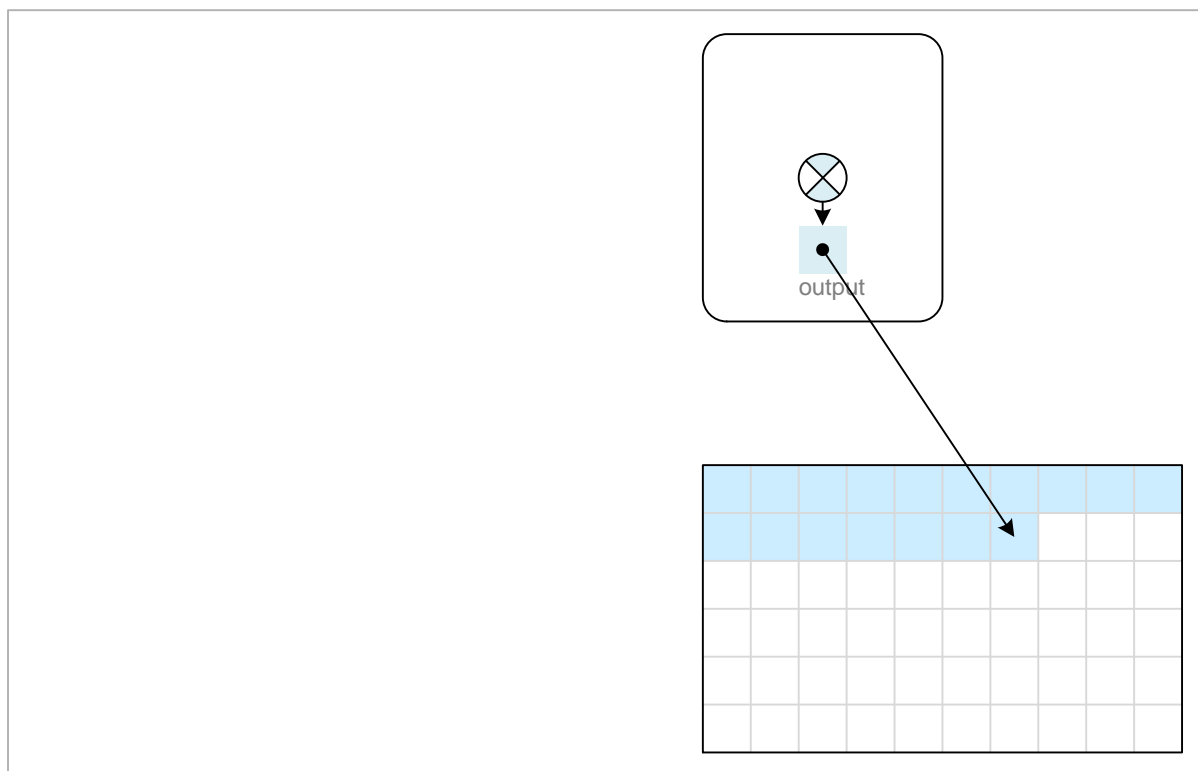
Common Operations

Fill

A fill is the simplest operation the M2MC can perform. It consists of writing a constant value to all pixels in a surface. The constant value is set as the “constant color” of the “blend” block of the M2MC. The “output” block is configured to take the color value directly from the blend block.

constant color: A constant value used by a block instead of reading a value from an input surface

output: The output block of the of the M2MC



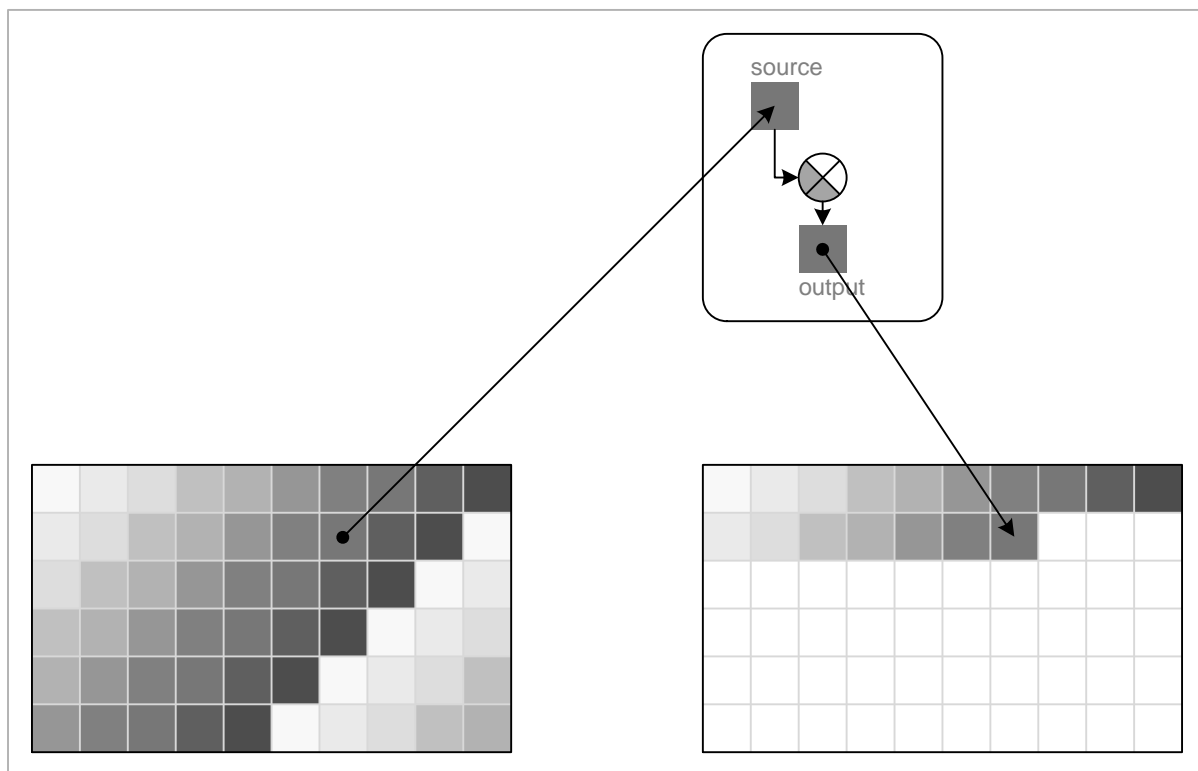
Fill Operation “in progress”

Copy

A copy is a basic operation that reads pixels from the source surface and writes them to the output surface. As with the fill, the output block is configured to take the color value from the blend block even though no blending is being performed. For a basic, unscaled copy, no other processing is performed on the pixel value.

Although this diagram shows an unscaled copy, the M2MC also supports performing scaling during the copy. The scaling occurs in the source block.

source: One of the two input blocks in the M2MC



Copy Operation "in progress"

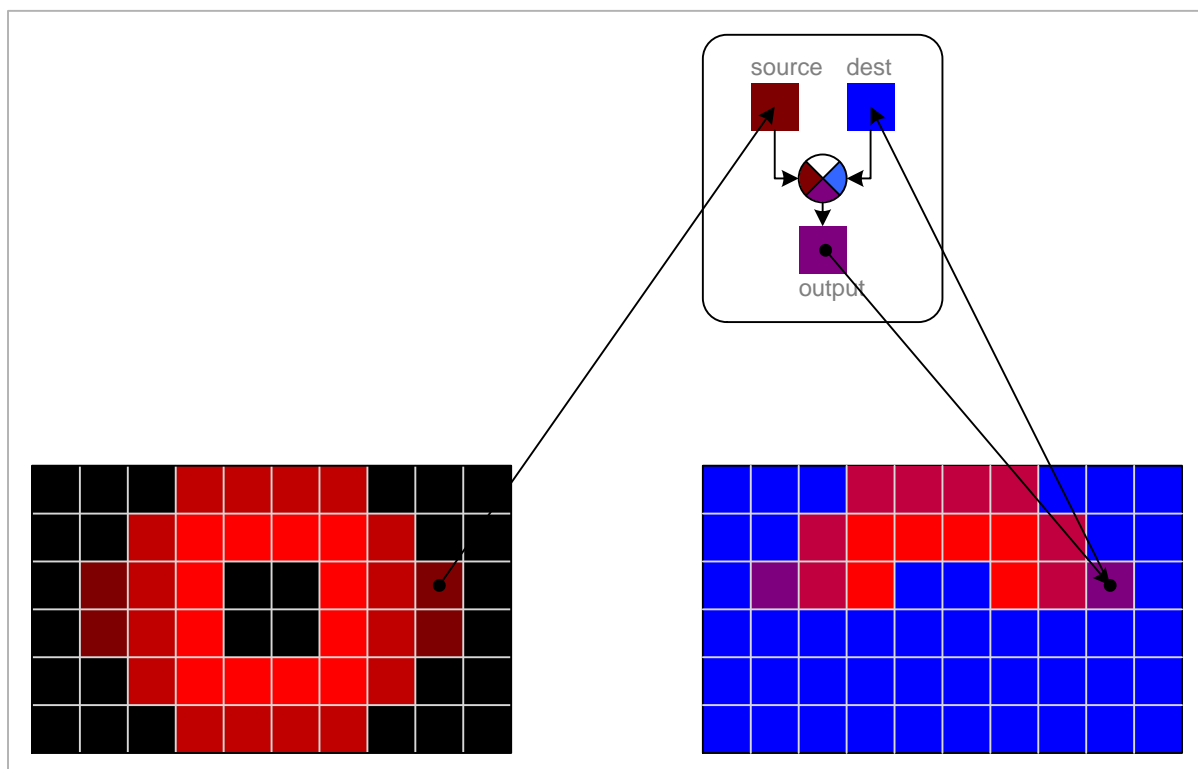
Blend

A blend operation reads from two inputs, combines the pixel value according to a “blending equation”, and writes the resulting pixel to the output. Typically, although not required, the output is also one of the inputs. This leads to the very unusual name of the second input: the “destination”. (This name pre-dates the M2MC and was adopted from the graphics industry.)

destination: The second of the two *input* blocks in the M2MC.

The blend operation is the most common use for the M2MC. It allows surfaces to be layered upon each other and allows smaller surfaces to be composited onto larger surfaces. The distinguishing factor of a blend is that the source surface pixels are combined with the destination (an *input*) surface pixels. Because these pixels are being combined, the M2MC can select how much of each pixel’s color to contribute to the output pixel, typically by using the alpha value of one or both of the pixels. The exact blend is specified by the “blending equation”.

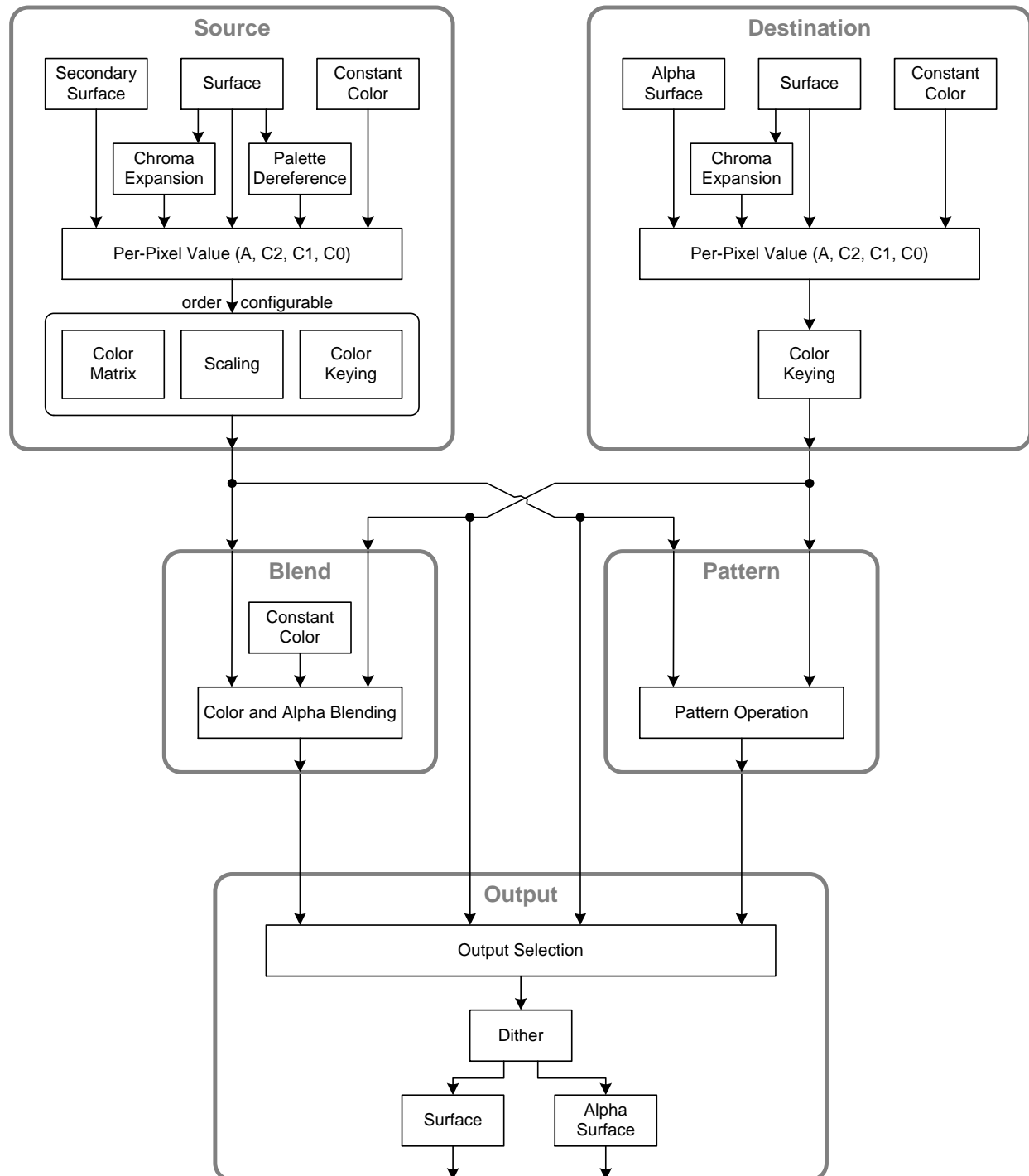
blending equation: An equation that specifies how to combine the source pixel value and the destination pixel value



Blend Operation “in progress”

M2MC Overview

The following diagram is an overview of the M2MC. The sections after provide details about each of the blocks in the M2MC. As a reminder, the destination block is an **input** block. The **output** is handled by the output block.

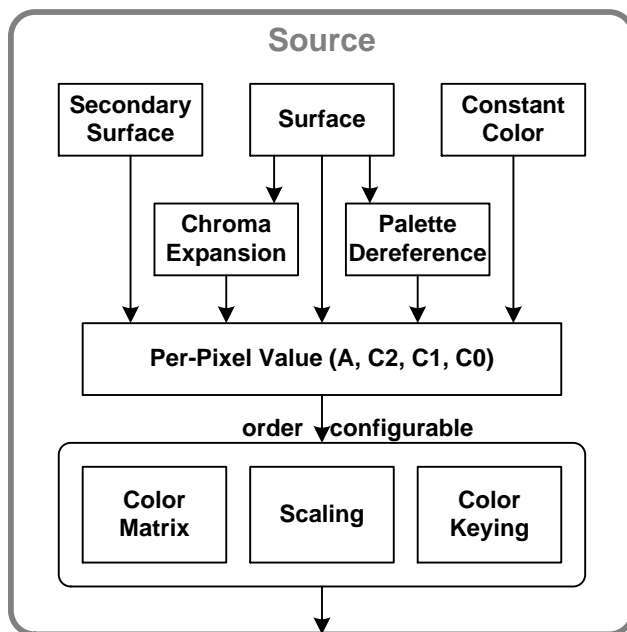


Internal Pixel Format

Internally, the M2MC operates on three color channels and an alpha channel. The M2MC is agnostic about the meaning of each of the color channels, which is why they are referred to as “C2, C1, and C0”. That is, they can be RGB, YUV, or a palette index. These are 8-bit channels.

The first step in each of the input blocks is to convert the pixels of the input into these four channels. (Note that it is possible to bypass the palette lookup; in this case a single color channel is used and it is the palette index. This is used for a fill or copy operation with a palette surface output.)

Source block

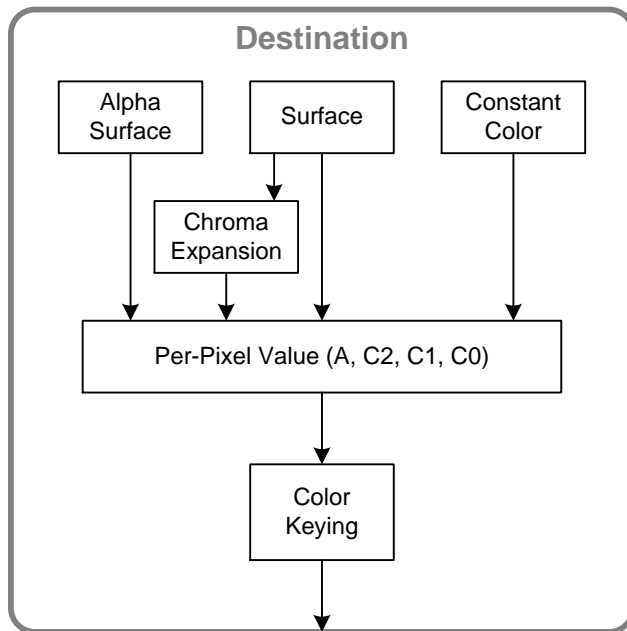


The source block delivers pixels from the source surface. Roughly speaking, this happens in two phases. The first phase converts the input pixels into the internal 8-bit per channel format, where the channels are referred to as A, C2, C1, and C0. If the surface is already 8-bit per channel, then no conversion is done. If the surface is less than 8-bits per channel, then the value of each channel is converted appropriately. If there is a constant color instead of a surface, then this is used for all colors. If the surface has a palette and palette dereference is specified, then the colors from the palette are used. Note that if the surface does not have all four channels, then the constant color provides the missing channel values.

Once the color is represented in the A, C2, C1, C0 format, then the color matrix, scaling and color keying can be performed. The color matrix is a 5x4 matrix that can perform an arbitrary matrix multiply with the four channels of the pixel color. A typical use is to convert from YUV colorspace to RGB or vice versa. This can also be used to swap color channels, drop color

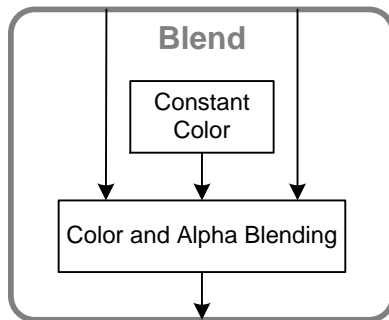
channels, or convert a color image to grayscale. The scaling capability can scale the source surface up or down and provides a multi-tap filter to allow for smooth scaled images. The color keying capability can replace selected colors in the source surface with alternate colors, either as a constant value or using pixels from the destination block. The order of these operations is configurable.

Destination block



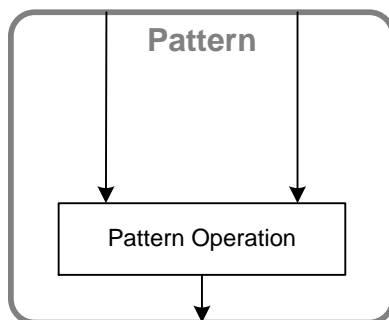
The destination (an input) block is simpler than the source block. It can not handle palette surfaces or perform scaling and does not have color conversion matrix. Except for colorkeying, the destination block simply provides unscaled pixels from a non-palette surface or from a constant color (or a combination of the two). Typically, although this is not required, the destination block is also the output block when performing blends. The Blend section under Common Operations shows how this would be used.

Blend block



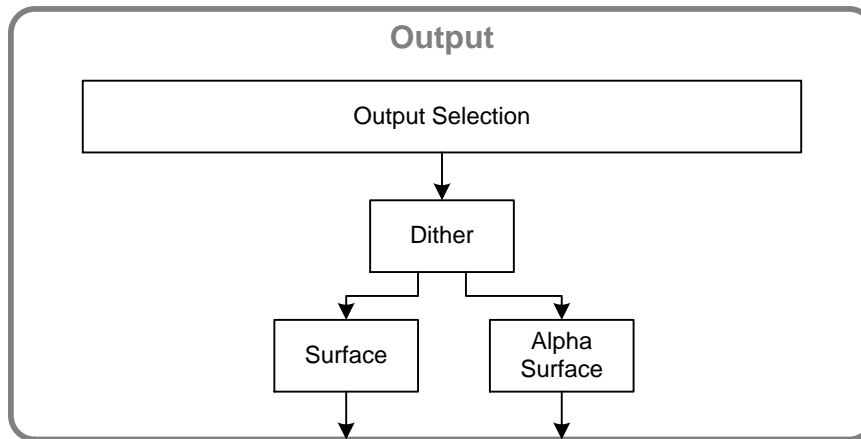
The blend block takes the inputs from the source block and the destination block and its own constant color value and combines these using the blend equation. The blend equation specifies how to compute a new color using these input values. This is described in detail in the “Blend Equations” section, below.

Pattern block



The Pattern block provides a “raster operation” (ROP) with the source and destination inputs. This feature is rarely used. Please contact Broadcom for more information.

Output block



The output block selects from any of four inputs (the source block, the destination block, the blend block, or the pattern block) and writes to the output surface.

Blend Equations

As mentioned above, the Blend block implements a blend equation that combines color inputs and generates a color output. Conceptually, it is a function that takes two pixel inputs and creates a pixel output:

$$f\left(\begin{array}{|c|c|c|c|} \hline \text{Input 1} & & & \\ \hline A & C0 & C1 & C2 \\ \hline \end{array} , \begin{array}{|c|c|c|c|} \hline \text{Input 2} & & & \\ \hline A & C0 & C1 & C2 \\ \hline \end{array} \right) \rightarrow \begin{array}{|c|c|c|c|} \hline \text{Output} & & & \\ \hline A & C0 & C1 & C2 \\ \hline \end{array}$$

Up until this point only a single “blend equation” has been implied, but, in fact, the blend block supports two blend equations: one for the alpha channel and one for the color channels. This allows for the alpha values to be computed differently from the color values, which is typically necessary in most blend functions. A more accurate depiction would be

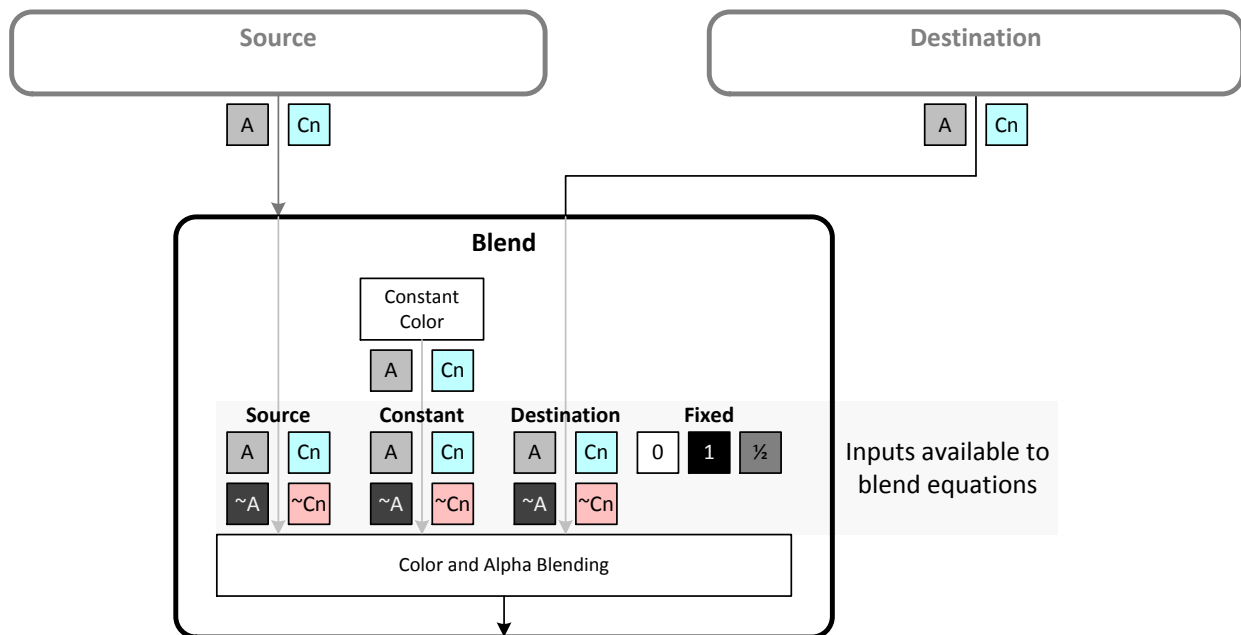
$$fa\left(\begin{array}{|c|} \hline \text{Input 1} \\ \hline A \\ \hline \end{array} , \begin{array}{|c|} \hline \text{Input 2} \\ \hline A \\ \hline \end{array} \right) \rightarrow \begin{array}{|c|} \hline \text{Output} \\ \hline A \\ \hline \end{array}$$

$$fc\left(\begin{array}{|c|} \hline \text{Input 1} \\ \hline Cn \\ \hline \end{array} , \begin{array}{|c|} \hline \text{Input 2} \\ \hline Cn \\ \hline \end{array} \right) \rightarrow \begin{array}{|c|} \hline \text{Output} \\ \hline Cn \\ \hline \end{array}$$

Note that each of the color channels are treated independently. It is not possible to combine values from different color channels. But, as expected, the alpha channels and the individual color channels can be combined to generate the output pixel.

Also note that the blend equations use the internal format-agnostic color format. The color channels may be Y, U, V or may be R, G, B, or they may be something else, as determined by the developer.

The inputs for the blend equations can come from the source block, the destination block, the constant color within the blend block, or can be fixed values (0, 1, ½). (Note that the source block and destination block also support constant colors.) The alpha component and color component can be selected independently. And the the inverse of any alpha input and any color input can also be selected. In total, there are fifteen possible inputs available to the blend equations.



The blend equations for both alpha and color components have the form

$$\left(\boxed{A} \times \boxed{B} \right) \pm \left(\boxed{C} \times \boxed{D} \right) \pm \boxed{E} \rightarrow \boxed{\text{Out}}$$

In addition to the five (A, B, C, D, E) inputs, it is also possible to specify whether C*D and whether E are added to or subtracted from the result.

The blend equation is typically written as

$$\text{Output} = A * B \pm C * D \pm E$$

This degree of flexibility allows many different blending operations to be implemented. A few are given below.

Blend Factors

Within the Nexus software an enumeration is used to identify the blend equation inputs:

```
typedef enum NEXUS_BlendFactor
{
    NEXUS_BlendFactor_eZero,           /* Zero */
    NEXUS_BlendFactor_eHalf,          /* 1/2 */
    NEXUS_BlendFactor_eOne,           /* One */

    NEXUS_BlendFactor_eSourceColor,    /* Color from source */
    NEXUS_BlendFactor_eInverseSourceColor, /* 1-color from source */
    NEXUS_BlendFactor_eSourceAlpha,    /* Alpha from source */
    NEXUS_BlendFactor_eInverseSourceAlpha, /* 1-alpha from source */

    NEXUS_BlendFactor_eDestinationColor, /* Color from destination */
    NEXUS_BlendFactor_eInverseDestinationColor, /* 1-color from destination */
    NEXUS_BlendFactor_eDestinationAlpha, /* Alpha from destination */
    NEXUS_BlendFactor_eInverseDestinationAlpha, /* 1-alpha from destination */

    NEXUS_BlendFactor_eConstantColor,  /* Color from blend color. */
    NEXUS_BlendFactor_eInverseConstantColor, /* 1-color from blend color */
    NEXUS_BlendFactor_eConstantAlpha,  /* Alpha from blend color. */
    NEXUS_BlendFactor_eInverseConstantAlpha, /* 1-alpha from blend color */

    NEXUS_BlendFactor_eMax
} NEXUS_BlendFactor;
```

There is an enumeration for each of the fifteen possible inputs.

In the example blending functions, below, the common prefix is omitted and these are shortened to “eZero”, “eHalf”, etc.

Example Blending Equations

Fill

If using the constant color of the blend block for a fill the blending equations would specify the constant alpha and color:

$$\text{Alpha} = \text{eConstantAlpha} * \text{eOne} + \text{eZero} * \text{eZero} + \text{eZero}$$

$$\text{Color} = \text{eConstantColor} * \text{eOne} + \text{eZero} * \text{eZero} + \text{eZero}$$

These simplify to

$$\text{Alpha} = \text{eConstantAlpha}$$

$$\text{Color} = \text{eConstantColor}$$

which clearly represents a fill operation, where the output is a constant value.

Copy Source

Copying from the source block to the output block can either go through the blend block or bypass it. If it goes through the blend block, then it would use this equation:

$$\text{Alpha} = \text{eSourceAlpha} * \text{eOne} + \text{eZero} * \text{eZero} + \text{eZero}$$

$$\text{Color} = \text{eSourceColor} * \text{eOne} + \text{eZero} * \text{eZero} + \text{eZero}$$

These simplify to

$$\text{Alpha} = \text{eSourceAlpha}$$

$$\text{Color} = \text{eSourceColor}$$

Copying from the destination could be done with a similar equation.

Blend Source Over Destination

The most common blending operation is referred to as “source over destination.” With this blend the entire source pixel is combined with the part of the destination “behind” or “under” the source pixel. The idea of “behind” or “under” is represented with “inverse source alpha”.

$$\text{Alpha} = \text{eSourceAlpha} * \text{eOne} + \text{eDestinationAlpha} * \text{eInverseSourceAlpha} + \text{eZero}$$

$$\text{Color} = \text{eSourceColor} * \text{eOne} + \text{eDestinationColor} * \text{eInverseSourceAlpha} + \text{eZero}$$

Visualizing Source Over Destination Blending

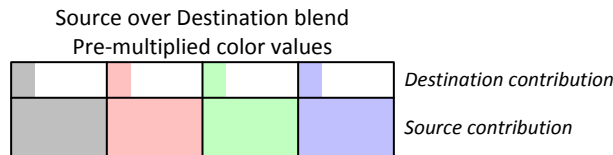
Below are representations of two ARGB pixels. Each channel is represented by a square. The area of one square represents a value of 1.0, the maximum value for the channel. For the color channels, the shaded area represents the range of values, from 0.0 to 1.0. For the alpha channels, the shaded area represents the actual alpha value. (In these diagrams, shading from the bottom or from the right has the same meaning and does not affect the representation of the value.)



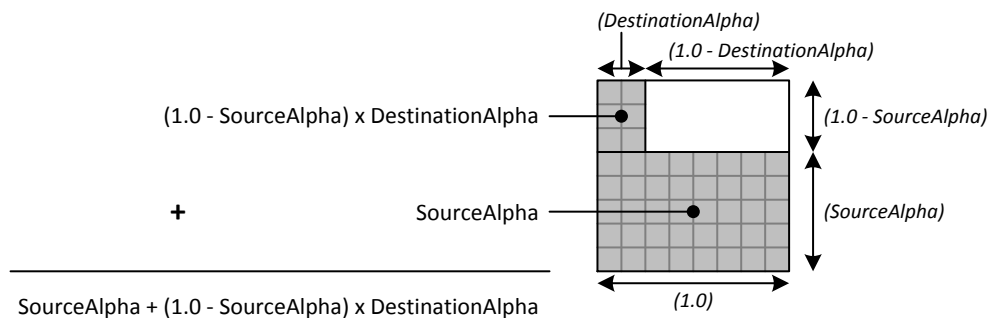
The color values for a pixel are always multiplied by the alpha of that pixel. The net effect is that the range of each color channel is constrained to be no more than the alpha value. This can be represented as follows.



For the most common blending operation between two surfaces, a Source Over Destination blend, the destination is “behind” the source and only the part (percentage) of the destination that is “not obscured” by the source contributes to the output pixel. Placing the source pixel representation over the destination pixel representation gives this diagram. Clearly the entire source pixel value contributes to the output. But only the part of the destination that “peeks through” the source contributes to the output.



For the alpha channel, the shaded area represents the new alpha value. For the new alpha value, the entire source alpha value is contributed plus that part of the destination alpha that is not obscured by the source. “Not obscured by the source” is equivalent to a multiplier of $(1.0 - \text{SourceAlpha})$ or, using the blending equation term, “inverse source alpha”.



With the input alpha values represented

$$\text{SourceAlpha} = 0.675 = 5/8,$$

$$\text{DestinationAlpha} = 0.25 = 2/8,$$

The output alpha is

$$5/8 + (1 - 5/8) \times 2/8 =$$

$$5/8 + 3/8 \times 2/8 =$$

$$40/64 + 6/64 =$$

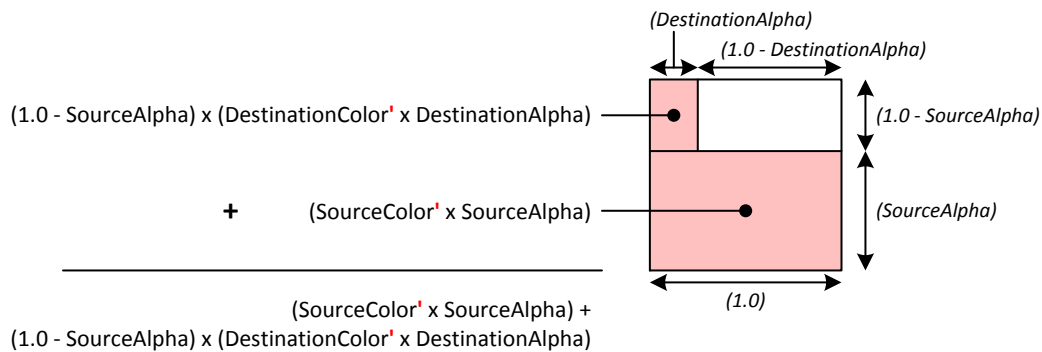
$$46/64$$

As a side note, the white area in the above has a value of $(1.0 - \text{SourceAlpha}) \times (1.0 - \text{DestinationAlpha})$. This would be the part that “shows through” for any future blending

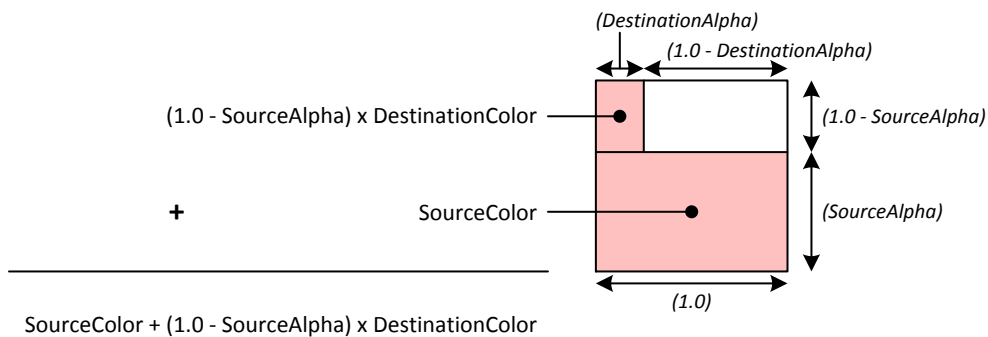
operation. Also, if either the SourceAlpha or the DestinationAlpha is 1.0, then the output pixel has an alpha of 1.0 and is opaque.

For the color channels, the discussion is slightly more complex, because there are the concepts of “original” or “full range” color channel values and of “constrained” or “alpha pre-multiplied” color channel values. The color channel representation shows the range of color channel values (from 0.0 to the maximum value now allowed by the alpha for each pixel) and not the actual color channel value (in contrast to the alpha channel, above).

When starting with original (unscaled) color channel values (indicated here by SourceColor^r and DestinationColor^r), the color value must be multiplied by the pixel’s alpha value to generate the correct contribution for that color. For example, if the alpha for a pixel is 0.5, then only 50% of the color will be in the output.



If the pixel’s color are already constrained, or pre-multiplied by the alpha of the pixel, where $\text{SourceColor} = \text{SourceColor}^r \times \text{SourceAlpha}$, then the representation simplifies to



This might seem like an unnecessary simplification that requires an extra step to “pre-compute” the surfaces, but it is actually a natural outcome of the blending. One of the observations that might be apparent in the above diagrams is that once a blending operation is performed, the output pixels are now pre-multiplied by the alpha of each pixel. In fact, it is traditionally difficult to generate a “full range” color channel value from a blend operation, because that requires a divide operation.

With alpha pre-multiplied input surfaces the source over destination blending equations are

$$\text{OutputAlpha} = \text{SourceAlpha} + (1.0 - \text{SourceAlpha}) \times \text{DestinationAlpha}$$

$$\text{OutputColor} = \text{SourceColor} + (1.0 - \text{SourceAlpha}) \times \text{DestinationColor}$$

As mentioned, these will generate an alpha pre-multiplied output surface.

Note that the M2MC can perform the pre-multiply step in both the source and destination blocks prior to delivering the pixels to the blend block.

Common API

The two sections following this describe two different APIs for interacting with the M2MC. This section describes the routines that can be used with both APIs. The complete description for these routines is given in the Nexus_Graphics.pdf document

Routine	Usage
NEXUS_Graphics2D_Checkpoint	Provides a mechanism to synchronize the software state with the hardware state.

Procedural API

The Nexus Graphics2D module provides both a procedural (conventional) API for performing graphics operations and a highly optimized “packet” API. The procedural API, listed in the table below, provides routines to perform most of the operations required of the M2MC. Refer to nexus_graphics2d.h for additional information.

Routine	Usage
NEXUS_Graphics2D_Fill	Fill a surface with a constant color. Provides color operations and allows the use of blend equations.
NEXUS_Graphics2D_Blut	Blit from one one or two surfaces to an output surface. This routine is highly configurable.
NEXUS_Graphics2D_PorterDuffFill	Fill routine that provides Porter-Duff operations with a fill (constant) color
NEXUS_Graphics2D_PorterDuffBlit	Blit routine that provides Porter-Duff operations
NEXUS_Graphics2D_DestripeBlit	(todo)

Packet Blit API

The 2D graphics core, the M2MC, is highly configurable. Analysis of the most common use cases found that typically only a few settings were changed between blits. A command pipeline design was adopted to optimize the application's interaction with the M2MC. The Nexus 2D graphics software architecture implements a command pipeline where small commands, "packets", are sent to the hardware. Each packet contains the settings for a small area of the M2MC core. Only the settings that are different between blits need to be sent; any common settings only need to be sent once.

Packet Pipeline Visualization

The Nexus graphics2d module maintains a packet pipeline. The next set of diagrams show how a scaled blit would be performed.

The first call by the application is to retrieve a pointer into the pipeline buffer

```
NEXUS_Graphics2D_GetPacketBuffer()
    application gets pointer into pipeline buffer
```

```
add BM2MC_PACKET_PacketSourceFeeder pPacket
```

```
add BM2MC_PACKET_PacketOutputFeeder pPacket
```

```
add BM2MC_PACKET_PacketBlend pPacket
```

```
add BM2MC_PACKET_PacketScaleBlit pPacket
```

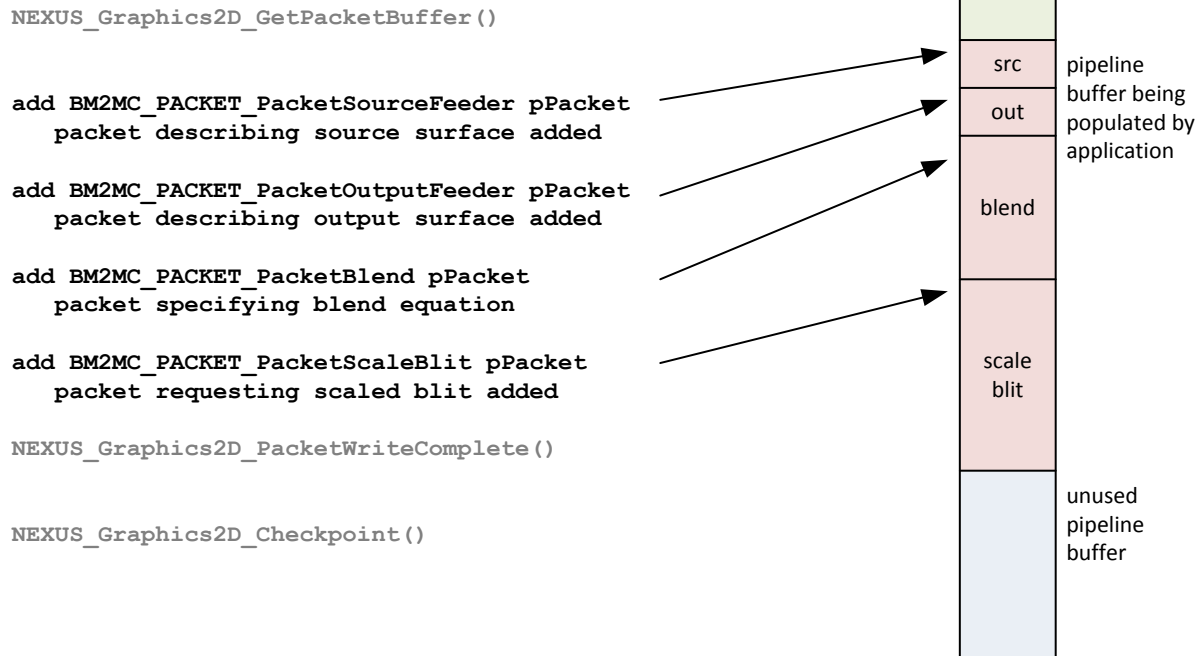
```
NEXUS_Graphics2D_PacketWriteComplete()
```

```
NEXUS_Graphics2D_Checkpoint()
```

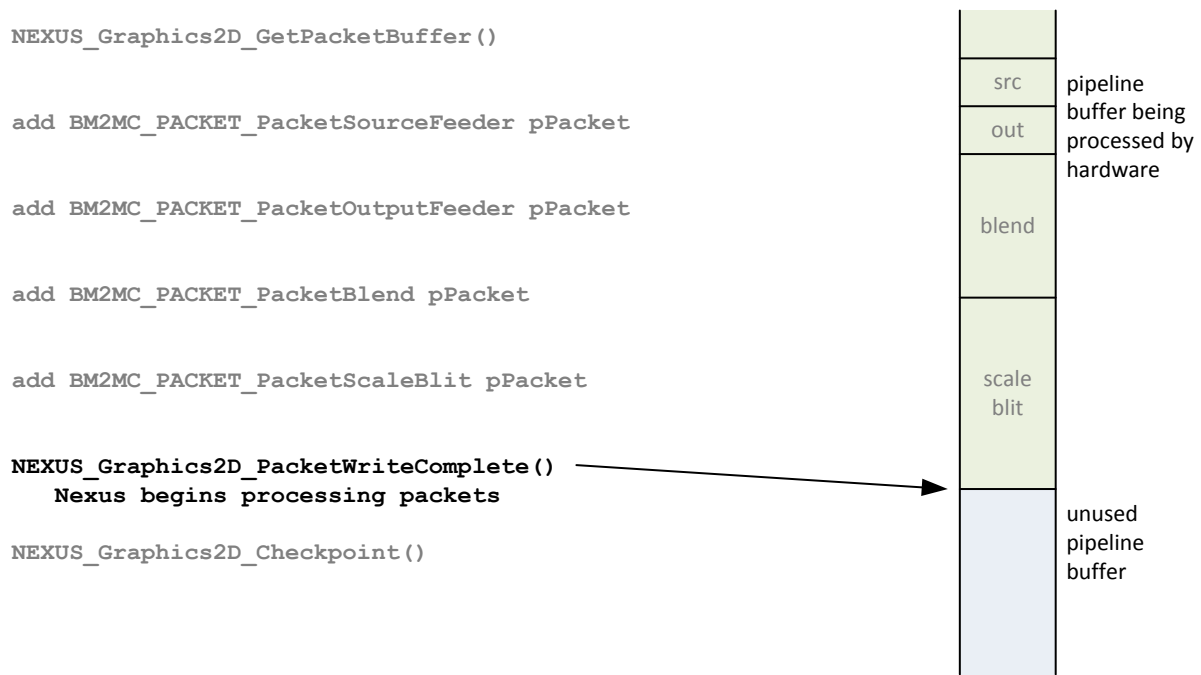


unused
pipeline
buffer

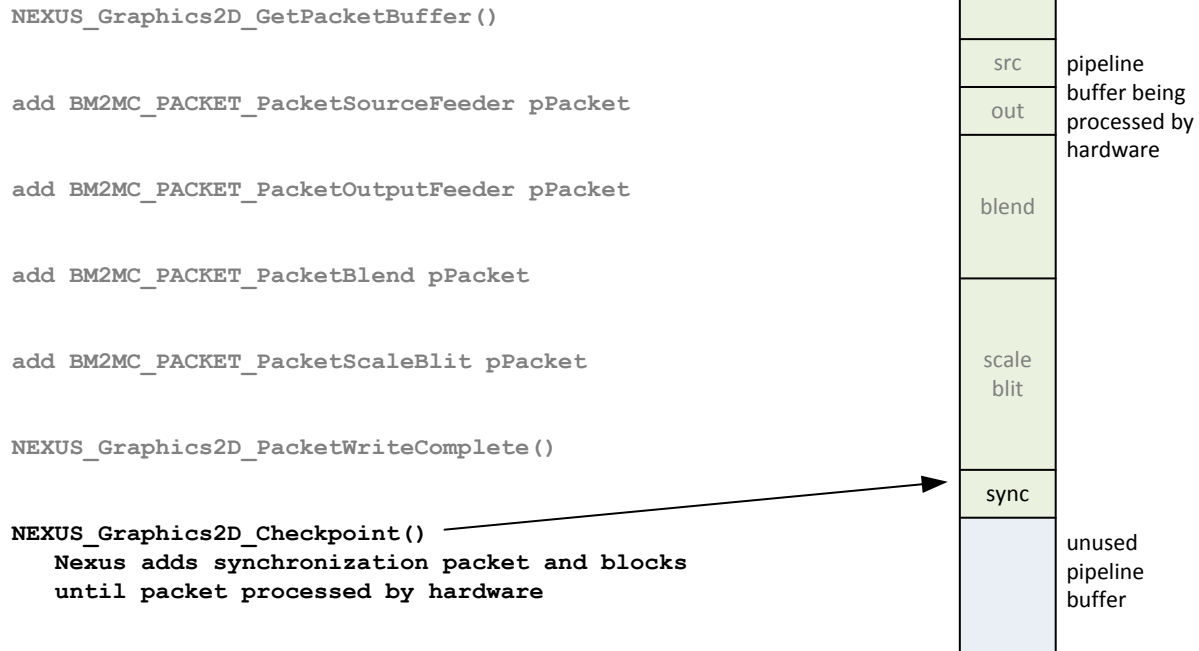
Once the application has access to the pipeline buffer, it adds blit packets



The application then tells Nexus that it has completed write packets. In response, Nexus will add the new packets to the list of packets being processed by Nexus and handed to the hardware.



Because the hardware runs asynchronously to the software, Nexus provides a mechanism to synchronize the software state with the hardware. This is only necessary at the points in the graphics composition when the application needs to know that the hardware has completed processing all of the graphics operations



Available Packet Types

Each packet is declared as a structure type and starts with “BM2MC_PACKET_”:

```

typedef struct
{
  BM2MC_PACKET_Header header;          /* packet header */
  BM2MC_PACKET_Blend color_blend;      /* color blend equation (defaults to eSourceColor) */
  BM2MC_PACKET_Blend alpha_blend;     /* alpha blend equation (defaults to eSourceAlpha) */
  uint32_t color;                     /* 32-bit color (A, R/Y, G/Cb, B/Cr/P) (defaults to 0) */
}
BM2MC_PACKET_PacketBlend;
  
```

The following is a list of packet types available in URSR 14.4. In the table the “BM2MC_PACKET_” prefix is removed for readability. (Note that deprecated packets are not listed.)

Packet Type	Usage
SourceFeeder	Set the surface in the source block
SourceFeeders	Set the primary and secondary surface (contact Broadcom for details)
StripedSourceFeeders	(todo: explain)
SourceColor	Set the constant color in the source block
SourceNone	Set the source to none (todo: what??)
SourceControl	Set additional controls within the source block
DestinationFeeder	Set the surface in the destination block

DestinationColor	Set the constant color in the destination block
DestinationNone	Set the destination to none (todo: explain)
DestinationControl	Set additional controls with the destination block
OutputFeeder	Set the surface in the output block
OutputCompression	Set the compression for the output (contact Broadcom)
OutputControl	Set additional controls within the output block
Blend	Set the blend equations and constant color in the blend block
BlendColor	Set the constant color in the blend block
Rop	Configure the ROP block
SourceColorkey	Set the source color key values
SourceColorkeyEnable	Enable source color key
DestinationColorkey	Set the destination color key values
DestinationColorkeyEnable	Enable the destination color key
Filter	Set the filter coefficients
FilterEnable	Enable the filter
FilterControl	Set additional filter controls, such as order of color key, filtering and matrix
SourceColorMatrix	Set the color conversion matrix values
SourceColorMatrixEnable	Enable thhe color conversion matrix
SourcePalette	Set the palette for the source surface
AlphaPremultiply	Enable premultiply of source color by source alpha
AlphaDemultiply	Enable “de-multiply” (divide) of blend color by blend alpha
DestAlphaPremultiply	Enable premultiply of destination color by destination alpha
Mirror	Enable the horizontal and/or vertical mirroring of source and/or destination
FillBlit	Request fill
CopyBlit	Request copy
BlendBlit	Request blend
ScaleBlit	Request scaled blit
ScaleBlendBlit	Request scaled blend
UpdateScaleBlit	Request scaled blend of partial (sub-rect) area
ResetState	Reset state of M2MC
SaveState	Save current state
RestoreState	Restore state

Coding Style

Broadcom uses the following coding style to add packets to the packet pipeline.

- The advancing buffer variable, “next”, is declared outside of the block where the next packet is declared.
- A new block is used for each packet, with braces to enclose the lines declaring the packet, populating the packet, and adding the packet to the buffer

- A pointer to a packet is declared using the packet type. A common variable name, in this case, “pPacket”, is used. This value is initialized with the advancing buffer variable, “next”.
- The packet is initialized with BM2MC_PACKET_INIT. Note that the second parameter is the packet type without the “BM2MC_PACKET_Packet” prefix.
- The fields of the packet are populated.
- The “next” variable is advanced with “++pPacket”. This uses the packet size, as determined by the compiler, to compute the next packet location.

This convention allows packets to added, removed, and rearranged without affecting the declarations elsewhere in the routine.

```
/* Note that using a new block (braces) around the use of each new packet type */
/* allows for a consistent usage style and variable naming */
{
    BM2MC_PACKET_PacketTypical *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, Typical, false);
    pPacket->param1 = value1;
    pPacket->param2 = value2;
    pPacket->param3 = value3;
    next = ++pPacket;
}
```

Sample Call Sequence

Working with the packet pipeline requires pointers to the start of the buffer, the next location to write into the buffer, and the size of the buffer.

```
NEXUS_Error rc;
void *buffer, *next;
size_t size;
```

If blend equations are needed, they can be declared as variables. This can be done in the scope of the routine or globally.

```
/* Color blend equation to copy color channels from constant color */
BM2MC_PACKET_Blend constantColorEq = {
    BM2MC_PACKET_BlendFactor_eConstantColor, BM2MC_PACKET_BlendFactor_eOne,
    false, BM2MC_PACKET_BlendFactor_eZero, BM2MC_PACKET_BlendFactor_eZero,
    false, BM2MC_PACKET_BlendFactor_eZero };

/* Alpha blend equation to copy alpha channel from constant color */
BM2MC_PACKET_Blend constantAlphaEq = {
    BM2MC_PACKET_BlendFactor_eConstantAlpha, BM2MC_PACKET_BlendFactor_eOne,
    false, BM2MC_PACKET_BlendFactor_eZero, BM2MC_PACKET_BlendFactor_eZero,
    false, BM2MC_PACKET_BlendFactor_eZero};
```

The first step is to get the pointer to the buffer and the size available. If the size is zero, then the internal queue is full. At this point the application typically waits until the pipeline has been processed.

```
rc = NEXUS_Graphics2D_GetPacketBuffer(gfx, &buffer, &size, 1024);
DBG_ASSERT(!rc);
if (!size) {
```

```

/* internal queue is full. wait for space to become available. */
BKNI_WaitForEvent(spaceAvailableEvent, BKNI_INFINITE);
continue;
}

next = buffer;

```

The source surface is specified with a SourceFeeder packet. Here, using the convention described above, the packet is declared, initialized, and populated, and the pointer is advanced.

```

{
    BM2MC_PACKET_PacketSourceFeeder *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, SourceFeeder, false);
    pPacket->plane = framebufferPlane;
    pPacket->color = 0;
    next = ++pPacket;
}

```

The output surface is specified with exactly the same sequence, except that the OutputFeeder packet is used.

```

{
    BM2MC_PACKET_PacketOutputFeeder *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, OutputFeeder, false);
    pPacket->plane = framebufferPlane;
    next = ++pPacket;
}

```

For many operation the blend equations need to be specified. That is done where with the Blend packet. Note that in this sample, the blend equations were declared earlier.

```

{
    BM2MC_PACKET_PacketBlend *pPacket = (BM2MC_PACKET_PacketBlend *)next;
    BM2MC_PACKET_INIT(pPacket, Blend, false);
    pPacket->color_blend = copyColorEq;
    pPacket->alpha_blend = copyAlphaEq;
    pPacket->color = 0;
    next = ++pPacket;
}

```

Add the ScaleBlit packet, to instruct the M2MC to scale from the source to the destination. Here the third parameter to BM2MC_PACKET_INIT macro is “true”, indicating the the M2MC should execute the blit after reading this packet.

```

{
    BM2MC_PACKET_PacketScaleBlit *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, ScaleBlit, true);
    pPacket->src_rect = src_rect;
    pPacket->out_rect = out_rect;
    next = ++pPacket;
}

```

Nexus needs to be notified once the packets have been added to the pipeline.

```

rc = NEXUS_Graphics2D_PacketWriteComplete(gfx, (uint8_t*)next - (uint8_t*)buffer);
BDBG_ASSERT(!rc);

```

If the application can call the Checkpoint routine if it needs to know that the graphics operations have completed. This would be done prior to giving the surface to the display. If not it is possible to see graphics artifacts.

```

rc = NEXUS_Graphics2D_Checkpoint(gfx, NULL);
if (rc == NEXUS_GRAPHICS2D_QUEUED) {
    BKNI_WaitForEvent(checkpointEvent, BKNI_INFINITE);
}

```

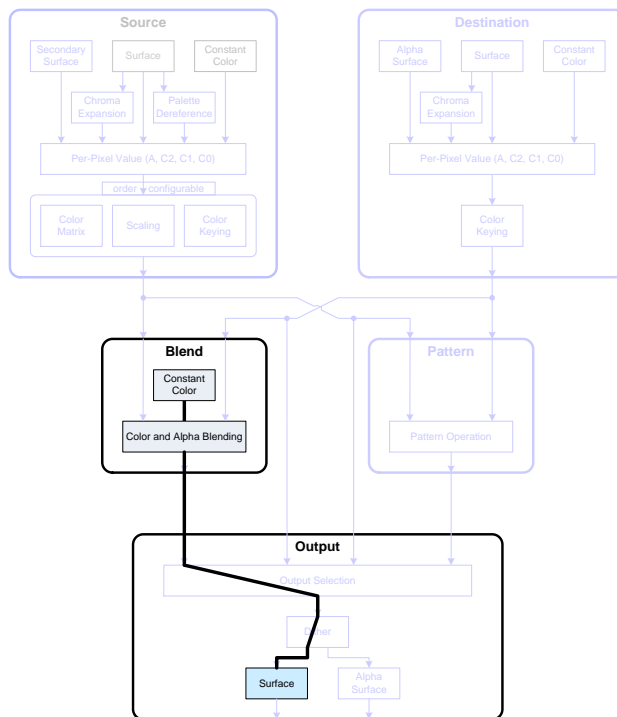
```
}
}
```

Configuring Common Operations

The following sections lists the packets needed to perform specific graphics operations using the packet blit API. Note that these are meant as samples of how to use different packet blit packet types. To efficiently use the packet blit API, the application could create a routine that incorporates all of the graphics operations needed to compose a scene or frame buffer.

Fill

A fill operation can be performed in at least three different ways: using the constant color in the source block, using the constant in the destination block, or using the constant color in the blend block. Here the blend block is used:



The simplified blend equations to use the constant color from the blend block are

$$\text{Alpha} = e\text{ConstantAlpha}$$

$$\text{Color} = e\text{ConstantColor}$$

These equation can be defined with the following. Note that these lines define the blend equations, but are not specifying them to the M2MC. That occurs below, in the Blend packet.

```

/* Color blend equation to copy color channels from constant color in blend block */
BM2MC_PACKET_Blend constantColorEq = {
    BM2MC_PACKET_BlendFactor_eConstantColor, BM2MC_PACKET_BlendFactor_eOne,
    false, BM2MC_PACKET_BlendFactor_eZero, BM2MC_PACKET_BlendFactor_eZero,
    false, BM2MC_PACKET_BlendFactor_eZero };

/* Alpha blend equation to copy alpha channel from constant color in blend block */
BM2MC_PACKET_Blend constantAlphaEq = {
    BM2MC_PACKET_BlendFactor_eConstantAlpha, BM2MC_PACKET_BlendFactor_eOne,
    false, BM2MC_PACKET_BlendFactor_eZero, BM2MC_PACKET_BlendFactor_eZero,
    false, BM2MC_PACKET_BlendFactor_eZero};

```

A packet for the surface to be filled, the output surface, is required.

```

{
    BM2MC_PACKET_PacketOutputFeeder *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, OutputFeeder, false);
    pPacket->plane = fillPlane;
    next = ++pPacket;
}

```

A packet to set the blend equations and the constant color is also required.

```

{
    BM2MC_PACKET_PacketBlend *pPacket = (BM2MC_PACKET_PacketBlend *)next;
    BM2MC_PACKET_INIT( pPacket, Blend, false );
    pPacket->color_blend = constantColorEq;
    pPacket->alpha_blend = constantAlphaEq;
    pPacket->color = fillColor;
    next = ++pPacket;
}

```

Finally the packet that identifies the operation, provides the rectangle on the output surface where the fill should occur, and sets the 'execute' bit (third paramter of BM2MC_PACKET_INIT) to true is required.

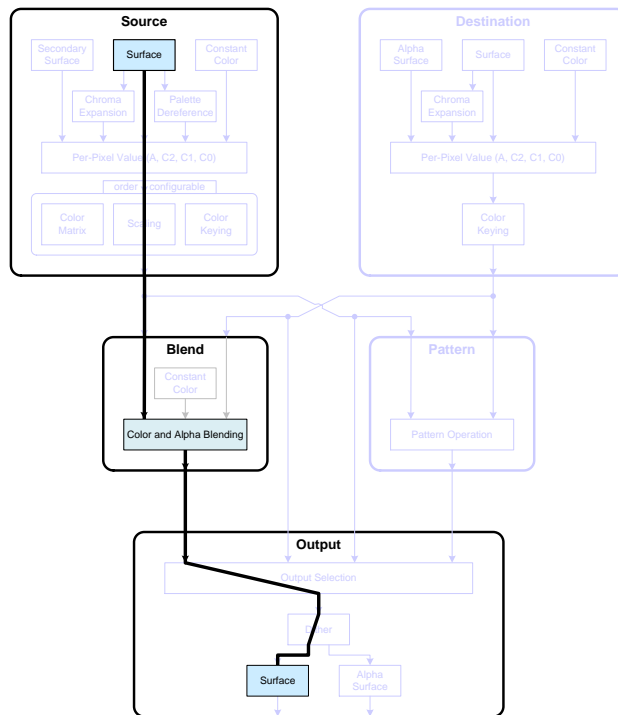
```

{
    BM2MC_PACKET_PacketFillBlit *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, FillBlit, true);
    pPacket->rect.x = 0;
    pPacket->rect.y = 0;
    pPacket->rect.width = createSettings.width;
    pPacket->rect.height = createSettings.height;
    next = ++pPacket;
}

```

Copy

A copy operation can go through either the blend block or bypass it. By convention, Broadcom typically implements copy operations that use the blend block. A basic, unscaled, unfiltered copy operation involves only four packets: one for each of the two surfaces, one for the blend equation, and one for the copy operation itself.



The simplified blend equations to use the source in the blend block are:

$$\text{Alpha} = \text{eConstantAlpha}$$

$$\text{Color} = \text{eConstantColor}$$

These equations are defined with these lines.

```
BM2MC_PACKET_Blend sourceColorEq = {
    BM2MC_PACKET_BlendFactor_eSourceColor, BM2MC_PACKET_BlendFactor_eOne,
    false, BM2MC_PACKET_BlendFactor_eZero, BM2MC_PACKET_BlendFactor_eZero,
    false, BM2MC_PACKET_BlendFactor_eZero};

BM2MC_PACKET_Blend sourceAlphaEq = {
    BM2MC_PACKET_BlendFactor_eSourceAlpha, BM2MC_PACKET_BlendFactor_eOne,
    false, BM2MC_PACKET_BlendFactor_eZero, BM2MC_PACKET_BlendFactor_eZero,
    false, BM2MC_PACKET_BlendFactor_eZero};
```

A packet for the output surface is required.

```
{
    BM2MC_PACKET_PacketOutputFeeder *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, OutputFeeder, false);
    pPacket->plane = outputPlane;
    next = ++pPacket;
}
```

And a packet for the source surface is also required.

```
{
    BM2MC_PACKET_PacketSourceFeeder *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, SourceFeeder, false);
    pPacket->plane = sourcePlane;
    pPacket->color = 0;
    next = ++pPacket;
}
```

```
}
```

A packet to set the blend equations is required. (Note that the color is not used, but only because of the blend equations do not reference the constant color or constant alpha.)

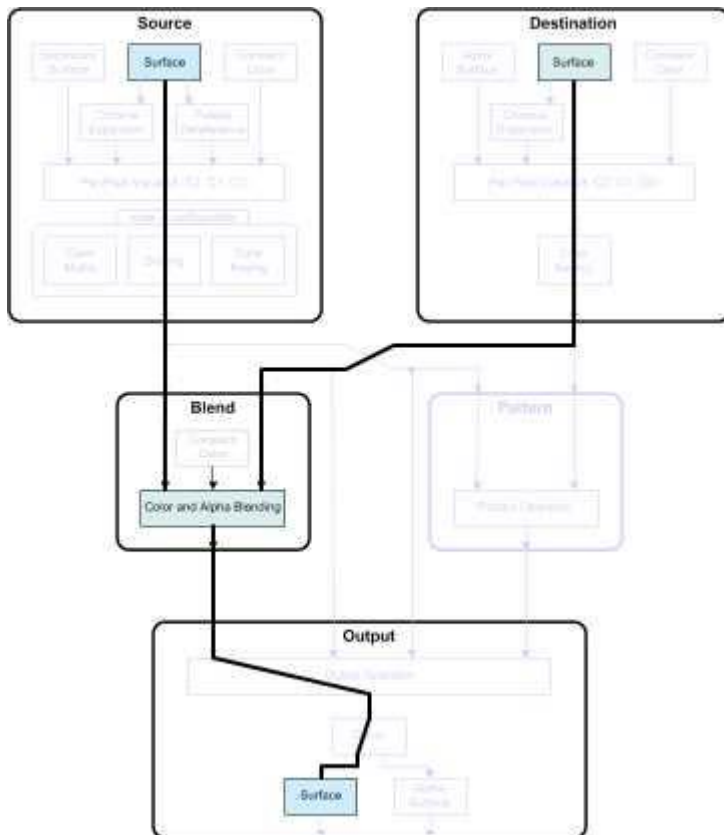
```
{
    BM2MC_PACKET_PacketBlend *pPacket = (BM2MC_PACKET_PacketBlend *)next;
    BM2MC_PACKET_INIT( pPacket, Blend, false );
    pPacket->color_blend = sourceColorEq;
    pPacket->alpha_blend = sourceAlphaEq;
    pPacket->color = 0;
    next = ++pPacket;
}
```

Finally a packet to instruct the M2MC to perform a (non-scaled) copy operation, which specifies the rectangle for the source, and the location of the rect in the output.

```
{
    BM2MC_PACKET_PacketCopyBlit *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, CopyBlit, true);
    pPacket->src_rect.x = sourceRect.x;
    pPacket->src_rect.y = sourceRect.y;
    pPacket->src_rect.width = sourceRect.width;
    pPacket->src_rect.height = sourceRect.height;
    pPacket->out_point.x = outX;
    pPacket->out_point.y = outY;
    next = ++pPacket;
}
```

Blend

A blend operation takes pixels from both the source and destination surfaces, combines them according to the blend equation, and writes the result to the output surface. The most common blend is called “source over destination”



These blend equations are defined by these lines. Note that both the source and destination colors and alphas are used and that the destination is multiplied by the inverse source alpha.

```
BM2MC_PACKET_Blend blendColorEq = {
    BM2MC_PACKET_BlendFactor_eSourceColor, BM2MC_PACKET_BlendFactor_eOne,
    false, BM2MC_PACKET_BlendFactor_eDestinationColor, BM2MC_PACKET_BlendFactor_eInverseSourceAlpha,
    false, BM2MC_PACKET_BlendFactor_eZero};

BM2MC_PACKET_Blend blendAlphaEq = {
    BM2MC_PACKET_BlendFactor_eSourceAlpha, BM2MC_PACKET_BlendFactor_eOne,
    false, BM2MC_PACKET_BlendFactor_eDestinationAlpha, BM2MC_PACKET_BlendFactor_eInverseSourceAlpha,
    false, BM2MC_PACKET_BlendFactor_eZero};
```

A packet for the output surface is required.

```
{
    BM2MC_PACKET_PacketOutputFeeder *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, OutputFeeder, false);
    pPacket->plane = outputPlane;
    next = ++pPacket;
}
```

A packet that specifies the destination surface is required. In this particular example, the output is also the destination, an input. Note that this is not required, but very typical.

```
{
    BM2MC_PACKET_PacketDestinationFeeder *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, DestinationFeeder, false);
    pPacket->plane = outputPlane;
    next = ++pPacket;
}
```

And a packet for the source surface is also required.

```
{
    BM2MC_PACKET_PacketSourceFeeder *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, SourceFeeder, false );
    pPacket->plane = sourcePlane;
    pPacket->color = 0;
    next = ++pPacket;
}
```

A packet to set the blend equations is required.

```
{
    BM2MC_PACKET_PacketBlend *pPacket = (BM2MC_PACKET_PacketBlend *)next;
    BM2MC_PACKET_INIT( pPacket, Blend, false );
    pPacket->color_blend = blendColorEq;
    pPacket->alpha_blend = blendAlphaEq;
    pPacket->color = 0;
    next = ++pPacket;
}
```












Finally a packet to instruct the M2MC to perform a (non-scaled) copy operation is required. This packet specifies the rectangle for the source, and the location of the rectangle in the destination and in the output.

```
{
    BM2MC_PACKET_PacketBlendBlit *pPacket = next;
    BM2MC_PACKET_INIT(pPacket, BlendBlit, true);
    pPacket->src_rect.x = sourceRect.x;
    pPacket->src_rect.y = sourceRect.y;
    pPacket->src_rect.width = sourceRect.width;
    pPacket->src_rect.height = sourceRect.height;
    pPacket->dst_point.x = outX;
    pPacket->dst_point.y = outY;
    pPacket->out_point.x = outX;
    pPacket->out_point.y = outY;
    next = ++pPacket;
}
```

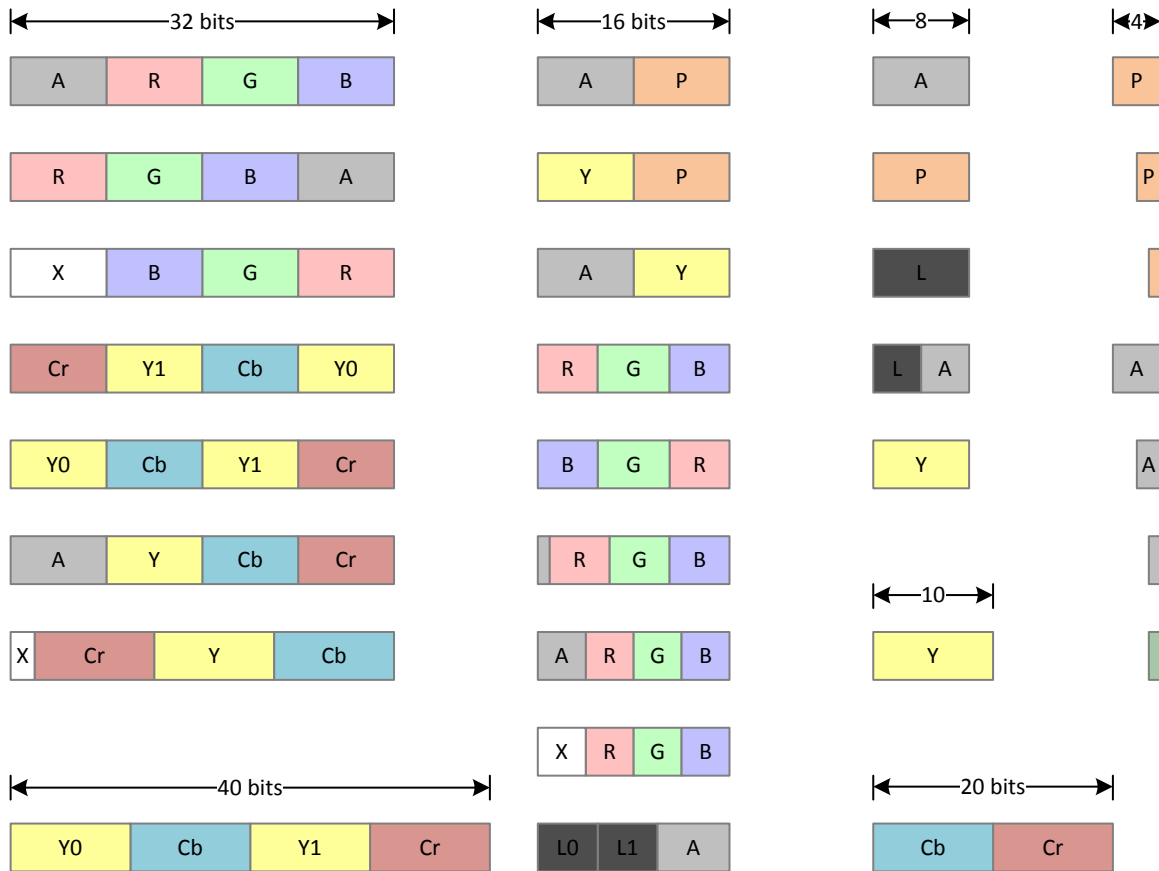

Appendix 1: Pixel Formats

The M2MC supports a wide ranges of pixel formats. The format is specified through a set of registers that allows for the complete customization of the format. This customization specifies the component type of each channel, the width (in bits) of each channel and the location of the channel within the pixel. Not all possible pixel formats can be handled by the M2MC, even if they can be defined. The most notable example is the 24 bit per pixel RGB format. Nexus provides a predefined set of pixel formats, which are listed in `nexus_types.h`.

The following represents the channel types that are supported:

 Alpha	 Red	 Y	 Luma
 Unused	 Green	 Cr	 W
 Palette	 Blue	 Cb	

These can be combined and ordered in many different ways. This diagram depicts *some* of the formats supported by Nexus. The bits per component of each channel is represented by the width in the diagram. Supported component widths are 1, 2, 4, 8, and 10 bits.



Additional information regarding YCbCr pixel formats:

- YCbCr444 means each sample has a Y, a Cb, and a Cr value, so Y:Cb:Cr = 1:1:1 in stream.
- YCbCr422 means for every two horizontal Y samples, there is one Cb and Cr sample, so Y:Cb:Cr = 2:1:1 in stream.
- 10-bit means each Y/Cb/Cr value is 10 bits, 8-bit means each Y/Cb/Cr value is 8 bits. For example, for YCbCr444 10bit, each sample has $Y(10\text{bit}) + Cb(10\text{bit}) + Cr(10\text{bit}) = 30$ bits and for YCbCr422 8bit, each sample has either $Y(8\text{bit}) + Cb(8\text{bit}) = 16$ bit or $Y(8\text{bit}) + Cr(8\text{bit}) = 16$ bits
- There is no pixel format for 4:2:0. The video decoder outputs 4:2:0 using separate luma (eY8) and chroma buffers (eCb8_Cr8), which is referred to as a “striped format.” See NEXUS_StripedSurfaceHandle.
- The 10-bit Y, Cr, Cb formats are typically produced by the video decoder core. The M2MC can read these formats, but it can not output them. Internally it converts these to the its 8-bit per channel format. This allows the M2MC to convert these specialized video formats into other formats that can then be combined with other graphics.

Appendix 2: Future Topics

- Definition of a surface.
- Color conversion matrix uses
- Pre-multiplied alpha
- Porter-Duff equations
- Secondary surfaces for input and output
- Output selection