REALmagic™

GFX Engine Reference

version 322

Oriol Prieto
Sigma Designs, Inc.

April 11, 2006

Contents

This standalone section is also part of SMP8630 software specification as 5.4.

1 Introduction to the GFX Engine

1.1 Capabilities

The GFX engine is a hardware graphics accelerator that your applications can use to:

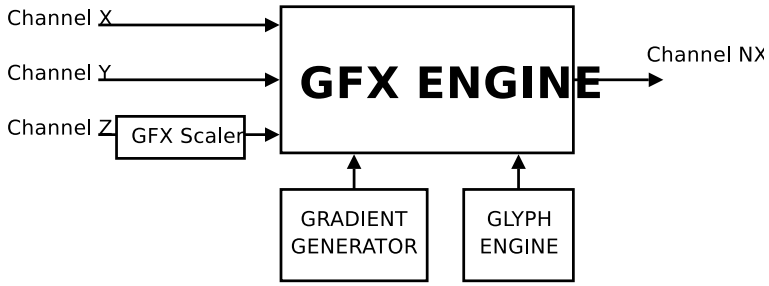
- Copy or alpha-blend a region of an image into another image. The region can be scaled and its color mode and format can be changed.
- Fill a rectangular region of an image with a given color. The fill can be solid or an alpha-blending.
- Draw 1bpp glyphs, including truetype character glyphs.
- Create or replace the alpha plane of an image.

On SMP8630 chips, the following capabilities are also available:

- Fill a region of an image with a radial or linear gradient. The fill can be solid or an alpha-blending.
- Draw glyphs, including truetype character glyphs, on any color mode supported by the GFXEngine.

1.2 Channels and Surfaces

The GFX engine has three input channels, called **X**, **Y**, **Z**, and one output channel called **NX**. Channels are the ports that the engine uses to read and write data to DRAM. Surfaces, on the other hand, can be seen as DRAM regions that contain an image.



Different commands may use different numbers of channels. Also, the set of channels that a given command uses may depend on its arguments. The Property Quick Reference below details which channels are used by every command. Before executing a command that uses a certain number of channels, you should have assigned a surface to each one of them, using the `Surface` property.
Some commands need a channel to be specified, via a `RMuint32`. You can then use one of the pre-defined constants:

- `GFX_SURFACE_ID_X`
- `GFX_SURFACE_ID_Y`
- `GFX_SURFACE_ID_Z`

- GFX_SURFACE_ID_NX

All the input channels can read from surfaces in indexed modes. Thus, a palette can be set to any of them. The output channel, on the other hand, should always be associated to a true color surface.

1.3 Tasks

While there is only one physical GFX Engine in your hardware, the software layer lets a number of applications use it concurrently. The EMhwlib will take care of switching contexts, if necessary, so that the applications can set up the engine freely without interfering with each other. This is achieved by means of what is called GFX Engine Tasks, and each application that wishes to use the GFX Engine should first create a Task. This is done via the `Open` property.

1.4 Using the GFX Multiscaler

Some of the commands use, besides the GFX Engine, the GFX Multiscaler. You will find information on wheter a command uses the GFX Multiscaler or not on the Property Quick Reference. Before using any of those those commands, your application should make sure that:

1. The scaler is on `Slave` mode on the mixers.
2. The scaler is not being used as a *deinterlacing companion* (deinterlacing type II).
3. The scaler is not being used a subpicture scaler.

2 GFXEngine Properties

2.1 DRAMSize

This *Exchange* property allows an application to know how much memory should be allocated for the GFX Engine buffers.

In Parameters: `[struct GFXEngine_DRAMSize_in_type]`

`[RMuint32] CommandFIFOCount`. All GFX Engine commands are sent to a command queue until they can be executed. This parameter allows you to set the length of that queue (trying to set a command when the queue is full will return the `RM_PENDING` value). A typical value for this parameter is 10.

Out Parameters: `[struct GFXEngine_DRAMSize_out_type]`

`[RMuint32] CachedSize` After the call, contains the size in bytes of the cached memory that needs to be allocated.

`[RMuint32] UncachedSize` After the call, contains the amount un uncached memory that needs to be allocated.

Hint: In order to allocate cached and uncached DRAM memory, you could write:

```
RUAMalloc(pRUA, 0, RUA_DRAM_CACHED, CachedSize);
RUAMalloc(pRUA, 0, RUA_DRAM_UNCACHED, UnCachedSize);
```

2.2 Open

Opens a GFX Engine Task. There is a maximum number of tasks, and each valid task index has a an associated `ModuleIndex` on the `GFXEngine` category. In order to know if a task index is free, and your application can acquire it, just try an open on its module; if the call succeeds it means that the task index has been reserved and initialized for your application. Use the `Enumerator`'s `CategoryIDToNumberOfInstances` property to obtain the max valid task index.
Note: Before opening a GFX Engine Task, you should have allocated the cached and uncached buffers. See the `DRAMSize` property.

Parameters: `[struct GFXEngine_Open_type]`

`[RMuint32] CommandFIFOCount` The maximum number of commands that can be queued, for this task. Set to the same value you used on the `DRAMSize` property.

`[RMuint32] Priority` Sets the priority of the new task. When commands from multiple tasks wait in the command queue, the task with the higher priority value will go first.

`[RMuint32] CachedAddress` Address of the cached buffer allocated for the task.

`[RMuint32] UnCachedAddress` Address of the uncached buffer allocated for the task.

`[RMuint32] CachedSize` Size, in bytes, of the cached buffer allocated for the task (see the `DRAMSize` property).

`[RMuint32] UncachedSize` Size, in bytes, of the uncached buffer allocated for the task (see the `DRAMSize` property).

2.3 Close

Frees a GFX Engine Task. Use this once your application is done using the GFX Engine.
Note: This function does not deallocate the cached or uncached buffers, that is left to the application.

Parameters: `[RMuint32]`
Set this field to 0.

2.4 CommandQueueEmpty

Lets the application monitor the state of the command queue.
In some cases the GFX commands need to be synchronized with other non accelerated operations. This typically implies waiting for all the queued commands to be finished before the non accelerated operations are performed.
The `CommandQueueEmpty`, unlike most other GFX commands, is a Get property.

Parameters: `[RMbool]` After the call, this is set to `TRUE` if the command queue is empty, or to `FALSE` if the task is not finished yet.

2.5 WaitForPicture

Writing on a region that is currently being displayed might cause tearing. The `WaitForPicture` command allows the `GFXEngine` to interact with the display in order to avoid this situation.
When this command is executed the `GFXEngine` will check wheter the specified picture is being displayed or not. If yes, processing of the command queue is suspended. The application can go on sending commands, that will just wait on the queue. As soon as the picture is no longer on display, processing of the pending commands goes on.

Parameters: `[RMuint32]` Adress to the blocking `struct EMhwlibPicture` we want to wait on.

2.6 DisplayPicture

Inserts the specified picture into a surface's picture fifo. If the surface is connected to an active scaler, this command actually allows to display the picture (please refer to the display chapter for more information on this). If the surface has an associated STC timer, a pts value must be provided. This pts will be used to schedule the displaying of the picture.

Parameters: `[struct GFXEngine_DisplayPicture_type]`

`[enum gfx_engine_surface_id] Picture` Address to the `struct EMhwlibPicture` to insert on the surface.

`[RMuint32] Surface` Address to the `struct EMhwlibSurface` to insert the picture into.

`[RMbool] Pts` Pts to be used by the display (only needed/used if the surface has an associated STC timer).

2.7 FlushCommandQueue

Processing of the command queue is interrupted and all remaining commands are eliminated. Waiting conditions (following a `WaitForPicture` command) are also cleared.

Note: This command is enqueued in just after the command being currently executed. Thus, it is not guaranteed that the queue will be empty right after `FlushCommandQueue`. It is guaranteed, however, that:

- Once it is sent, no commands previously enqueued will *start* its execution.
- All commands enqueued after `FlushCommandQueue` will be processed normally.

2.8 Surface

Specifies the position and shape in DRAM of a surface (image), and sets it as input/output of a given channel. For every surface, you should also specify its format using the `ColorFormat` command.

Parameters: `[struct GFXEngine_Open_type]`

`[enum gfx_engine_surface_id] SurfaceID` Specifies which channel the surface should be associated to.

`[RMuint32] TotalWidth` Width of the image. Note that this is the *total* length of the image's lines.

`[RMbool] Tiled` Reserved field. Set this to `FALSE`.

`[RMuint32] StartAddress` Address of the first pixel of the surface (on non-interleaved YUV surfaces, address of the first pixels of the luma plane).

`[RMuint32] ChromaStartAddress` For non-interleaved YUV surfaces, address of the first pixel of the chroma plane. Unused (can be left unset) if the surface's colormode is not `EMhwlibColorMode_VideoNonInterlea`

2.9 ColorFormat

Specifies the color format and color mode of the surface associated to a given channel. For indexed color modes, you should set up the palette via the `Palette_XBPP` command.

Parameters: `[struct GFXEngine_ColorFormat_type]`

`[enum gfx_engine_surface_id] SurfaceID` Specifies which channel the command should be applied to.

`[enum EMhwlibColorMode] MainMode` Specifies the color mode of the surface.
Note: On chips before 8630, color modes `EMhwlibColorMode_VideoInterleaved` and `EMhwlibColorMode_Videol` are only supported for channel **Z**.

`[enum EMhwlibColorFormat] SubMode` Specifies the color format of the surface.

`[enum EMhwlibSamplingMode] SamplingMode` Specifies the sampling mode of the surface.

`[enum EMhwlibColorSpace] ColorSpace` Specifies the sampling mode of the surface.

2.10 Palette_XBPP

This section covers the commands `Palette_1BPP`, `Palette_2BPP`, `Palette_4BPP`, `Palette_8BPP`. If one of the input channels is reading from an indexed surface, this command sets the palette to use.

Parameters: `[struct GFXEngine_Palette_XBPP_type]`

`[enum gfx_engine_surface_id] SurfaceID` Specifies which channel the command should be applied to. Note that the **NX** channel is not a valid value (it only supports true color output).

`[RMpalette_XBPP] Palette` The palette to set.

2.11 AlphaFormat

If the **X** channel is going to provide the alpha information on a `MoveRectangle`, `ReplaceRectangle`, `MoveAndScaleRectangl` or `ReplaceAndScaleRectangle`, this command specifies the format on which alpha information is stored on DRAM.

Parameters: `[struct GFXEngine_AlphaFormat_type]`

`[enum gfx_alpha_format] AlphaFormat` Specifies the format in which alpha information is stored on DRAM. The alpha information required by the GFX Engine is 8 bits per pixel. For modes `GFX_ALPHA_FORMAT_LUT_1BPA` and `GFX_ALPHA_FORMAT_LUT_2BPA` the input pixels are used to index a lookup table that can be setup via the `AlphaPalette` commmand. For mode `GFX_ALPHA_FORMAT_TRUE_4BPA` the 4bpp pixels are expanded to 8bpp using the expression:

$$Alpha_{out} = 17 * Alpha_{in}$$

2.12 AlphaPalette

Sets the lookup table that is used to decode indexed alpha values on a given channel.

Parameters: `[struct GFXEngine_AlphaPalette_type]`

`[enum gfx_engine_surface_id] SurfaceID` Species on which channel the palette should be set.

`[RMuint8] Alpha0` 8bit alpha value for palette entry 0.

`[RMuint8] Alpha1` 8bit alpha value for palette entry 1.

`[RMuint8] Alpha2` 8bit alpha value for palette entry 2.

`[RMuint8] Alpha3` 8bit alpha value for palette entry 3.

2.13 EnableAlphaFading

Enables or disables alpha fading on channel **Z**.
When 32bpp or 16bpp true color data is read through channel **Z**, the alpha values of the pixels can be modified by the `GFXEngine`. The applied transformation is:

$$Alpha_{out} = Alpha_{in} * Alpha1 + (1 - Alpha_{in}) * Alpha0$$

Where $Alpha_{out}$ is the resulting alpha value, $Alpha_{in}$ the normalized original alpha value. $Alpha0$, $Alpha1$ are obtained from the **Z** channel's alpha palette entries 0 and 1, respectively (see command `AlphaPalette`).

Parameters:

`[RMbool] Enable` `TRUE` enables alpha-fading (the alpha value of all pixels read through the **Z** channel are modified). `FALSE` disables alpha-fading.

2.14 KeyColor

If one of the input surfaces is set to mode `EMhwlibColorMode_TrueColorWithKey`, this command set its keycolor range.

Parameters: `[struct GFXEngine_KeyColor_type]`

`[enum gfx_engine_surface_id] SurfaceID` Specifies which channel the command should be applied to.

`[RMuint32] Color` First color on the keycolor range.

`[RMuint8] Range` Sets the width of the keycolor range.

The keycolor range is defined as:

$$keycolor\_range = [CB_i(C), CB_i(C) + 2^{range}[$$

Where C is the first color on the keycolor range, and CB_i represents each one of the three color components. If an input pixel falls into this range, the accelerator will treat it as a zero-value (transparent black) pixel.

2.15 FillRectangle

Fills a rectangle with a solid color on surface **NX**.
Note: The fill on this command is solid, i.e. all the pixels on the destination surface will be overwritten (see the Output description, below). If you want an alpha-blended fill, use `SingleColorBlendRectangles` instead.

Parameters: `[struct GFXEngine_FillRectangle_type]`

`[RMuint32] X` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface NX.

`[RMuint32] Y` Vertical offset of the top-left border of the rectangle, relative to the origin of surface NX.

`[RMuint32] Width` Width of the filled rectangle.

`[RMuint32] Height` Height of the filled rectangle.

`[RMuint32] Color` Fill color, in 32bpp ARGB format.

Output: All the components of a pixel on the output rectangle, Pix_{NX} are copied from the specified color, C.

$$\begin{aligned} Alpha(Pix_{NX}) &= Alpha(C) \\ CB_i(Pix_{NX}) &= CB_i(C) \end{aligned}$$

Where CB_i represents each one of the three color components.

2.16 BlendRectangles

Does an alpha-blending of two rectangles read from **Y** and **X**, and writes the resulting rectangle on the **NX** surface. Surface **Y** provides the alpha information for the blend (see the Output description below). All rectangles should have the same size (use the `BlendAndScaleRectangles` if you need scaling).

Parameters: `[struct GFXEngine_BlendRectangles_type]`

`[RMuint32] Src1X` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **Y**.

`[RMuint32] Src1Y` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **Y**.

`[RMuint32] Src2X` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **X**.

`[RMuint32] Src2Y` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **X**.

`[RMuint32] DstX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32] DstY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32] Width` Width of the blended rectangle.

`[RMuint32] Height` Height of the blended rectangle.

`[RMbool] SaturateAlpha` If set to `TRUE`, the alpha value of the resulting rectangle is set to `0xFF` for all pixels.

Output: The value of a pixel on the output rectangle, Pix_{NX} is calculated from the two input pixels, Pix_X and Pix_Y using the following formulas:

$$\begin{aligned} &Alpha(Pix_{NX}) = 255 && \text{If } \texttt{SaturateAlpha} = \texttt{TRUE} \\ &Alpha(Pix_{NX}) = \tfrac{Alpha(Pix_Y)}{255} + \tfrac{Alpha(Pix_X)}{255} - \tfrac{Alpha(Pix_Y)}{255} * \tfrac{Alpha(Pix_X)}{255} && \text{If } \texttt{SaturateAlpha} = \texttt{FALSE} \end{aligned}$$

$$CB_i(Pix_{NX}) = \frac{1}{255} * (CB_i(Pix_Y) * Alpha(Pix_Y) + CB_i(Pix_X) * (255 - Alpha(Pix_Y)))$$

Where CB_i represents each one of the three color components.

2.17 SingleColorBlendRectangles

Does an alpha blending of a specified color with a rectangle read from surface **X**, and writes the resulting rectangle on the **NX** surface. The specified color provides the alpha information for the blend (see below, on the Output section for this command).
Note: Executing this will modify the registers of the GFX MultiScaler (you don't need to associate a surface with the **Z** channel, though).

Parameters: `[struct GFXEngine_SingleColorBlendRectangles_type]`

`[RMuint32] Color` Color to use on the blend, in 32bpp ARGB format.

`[RMuint32] SrcX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **X**.

`[RMuint32] SrcY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **X**.

`[RMuint32] DstX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32] DstY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32] Width` Width of the blended rectangle.

`[RMuint32] Height` Height of the blended rectangle.

`[RMbool] SaturateAlpha` If set to `TRUE`, the alpha value of the resulting rectangle is set to `0xFF` for all pixels.

Output: The value of a pixel on the output rectangle, Pix_{NX} is calculated from the specified color, C and the input pixel Pix_Y using the same formulas than for the `BlendRectangles`.

2.18 MoveRectangle

Move a rectangle read from surface **Y**, to surface **NX**. Optionally, merge with alpha information of a rectangle read through the **X** channel. All rectangles should have the same size (use the `MoveAndScaleRectangles` property if you need scaling).

Parameters: `[struct GFXEngine_MoveReplaceRectangle_type]`

`[RMuint32] SrcX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **Y**.

`[RMuint32] SrcY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **Y**.

`[RMuint32] AlphaX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **X**.

`[RMuint32] AlphaY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **X**.

`[RMuint32] DstX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32] DstY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32] Width` Width of the moved rectangle.

`[RMuint32] Height` Height of the moved rectangle.

`[enum gfx_merge_mode] Merge`

`GFX_MERGE_MODE_DISABLE` The alpha information from the source (surface **Y**) is unmodified.

`GFX_MERGE_MODE_X` The **Y** surface provides only the color information, and the alpha information is obtained from surface **X**.

`GFX_MERGE_MODE_MODULATE` The alpha information from channels **X** and **Y** is combined. *Note:* This mode is available only on SMP863X chips.

Output: The value of a pixel on the output rectangle, Pix_{NX} is calculated from the two input pixels, $Alpha_X$ and Pix_Y using the following formulas:

$$
\begin{array}{ll}
Alpha(Pix_{NX}) = Alpha(Pix_Y) & \text{for } \texttt{GFX\_MERGE\_MODE\_DISABLE} \\
Alpha(Pix_{NX}) = Alpha_X & \text{for } \texttt{GFX\_MERGE\_MODE\_X} \\
Alpha(Pix_{NX}) = \frac{Alpha_X * Alpha(Pix_Y)}{256} & \text{for } \texttt{GFX\_MERGE\_MODE\_MODULATE}
\end{array}
$$

$$CB_i(Pix_{NX}) \quad = \quad CB_i(Pix_Y)$$

Where CB_i represents each one of the three color components.

2.19 ReplaceRectangle

Move a rectangle read from surface **Y**, to surface **NX**, but *only for pixels whose alpha value is greater than 0*. Optionally, merge with alpha information of a rectangle read through the **X** channel. All rectangles should have the same size (use the `ReplaceAndScaleRectangles` property if you need scaling).

Parameters: `[struct GFXEngine_MoveReplaceRectangle_type]`

`[RMuint32] SrcX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **Y**.

`[RMuint32] SrcY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **Y**.

`[RMuint32] AlphaX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **X**.

`[RMuint32] AlphaY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **X**.

`[RMuint32] DstX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32] DstY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32] Width` Width of the replaced rectangle.

`[RMuint32] Height` Height of the replaced rectangle.

`[enum gfx_merge_mode] Merge`

`GFX_MERGE_MODE_DISABLE` The alpha information from the source (surface **Y**) is unmodified.

`GFX_MERGE_MODE_X` The **X** surface provides only the color information, and the alpha information is obtained from surface **X**.

`GFX_MERGE_MODE_MODULATE` The alpha information from channels **X** and **Y** is combined. *Note:* This mode is available only on SMP8630 chips.

Output: The value of a pixel on the output rectangle, Pix_{NX} is calculated from the two input pixels, $Alpha_X$ and Pix_Y using the following formulas:

$$\begin{array}{ll} alpha\_value = Alpha(Pix_Y) & \text{for GFX\_MERGE\_MODE\_DISABLE} \\ alpha\_value = Alpha_X & \text{for GFX\_MERGE\_MODE\_X} \\ alpha\_value = \frac{Alpha_X * Alpha(Pix_Y)}{256} & \text{for GFX\_MERGE\_MODE\_MODULATE} \end{array}$$

For every pixel, if $alpha\_value > 0$

$$\begin{array}{rcl} Alpha(Pix_{NX}) & = & alpha\_value \\ CB_i(Pix_{NX}) & = & CB_i(Pix_Y) \end{array}$$

Where CB_i represents each one of the three color components.
Otherwise (if $alpha\_value = 0$) the pixel on the **NX** surface is unmodified.

2.20 BlendAndScaleRectangles

Does an alpha-blending of two rectangles read from **Z** and **X**, and writes the resulting rectangle on the **NX** surface. Surface **Z** provides the alpha information for the blend (see below, on the Output section for this command). The rectangle read through **Z** does not need to have the same size than the output rectangle.
Note: Executing this will modify the registers of the GFX MultiScaler.

Parameters: `[struct GFXEngine_BlendAndScaleRectangles_type]`

[RMuint32] Src1X Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **Z**.

[RMuint32] Src1Y Vertical offset of the top-left border of the rectangle, relative to the origin of surface **Z**.

[RMuint32] Src2X Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **X**.

[RMuint32] Src2Y Vertical offset of the top-left border of the rectangle, relative to the origin of surface **X**.

[RMuint32] DstX Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

[RMuint32] DstY Vertical offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

[RMuint32] SrcWidth Width of the rectangle read through **Z**.

[RMuint32] SrcHeight Height of the rectangle read through **Z**.

[RMuint32] DstWidth Width of the output rectangle (on surface **NX**). Also the width of the second input rectangle (read through surface **X**).

[RMuint32] DstHeight Height of the output rectangle (on surface **NX**). Also the height of the second input rectangle (read through surface **X**).

[RMbool] SaturateAlpha If set to TRUE, the alpha value of the resulting rectangle is set to 0xFF for all pixels.

Output: If the sizes of the source rectangle and the destination rectangle differ, the data read through **Z** is scaled to match the output size. Once that is done, every pixel on the output rectangle has one pixel from **Z** and one pixel from **X** associated, and the resulting output values are computed exactly as in the BlendRectangles command.

2.21 MoveAndScaleRectangles

Move a rectangle read from surface **Z**, to surface **NX**. Optionally, merge with alpha information of a rectangle read through the **X** channel. The rectangle read through **Z** does not need to have the same size than the output rectangle.
Note: Executing this will modify the registers of the GFX MultiScaler.

Parameters: [struct GFXEngine_MoveReplaceScaleRectangle_type]

[RMuint32] SrcX Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **Z**.

[RMuint32] SrcY Vertical offset of the top-left border of the rectangle, relative to the origin of surface **Z**.

[RMuint32] AlphaX Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **X**.

[RMuint32] AlphaY Vertical offset of the top-left border of the rectangle, relative to the origin of surface **X**.

[RMuint32] DstX Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

[RMuint32] DstY Vertical offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

[RMuint32] SrcWidth Width of the rectangle read through **Z**.

[RMuint32] SrcHeight Height of the rectangle read through **Z**.

[RMuint32] DstWidth Width of the output rectangle (on surface **NX**). Also the width of the alpha input rectangle (read through surface **X**.

[RMuint32] DstHeight Height of the output rectangle (on surface **NX**). Also the height of the alpha input rectangle (read through surface **X**.

[enum gfx_merge_mode] Merge

GFX_MERGE_MODE_DISABLE The alpha information from the source (surface **Z**) is unmodified.

GFX_MERGE_MODE_X The **X** surface provides only the color information, and the alpha information is obtained from surface **X**.

GFX_MERGE_MODE_MODULATE The alpha information from channels **X** and **Z** is combined. *Note:* This mode is available only on SMP8630 chips.

Output: If the sizes of the source rectangle and the destination rectangle differ, the data read through **Z** is scaled to match the output size. Once that is done, every pixel on the output rectangle has one pixel from **Z** and one pixel from **X** associated, and the resulting output values are computed exactly as in the `MoveRectangle` command.

2.22 ReplaceAndScaleRectangles

Move a rectangle read from surface **Z**, to surface **NX**, but*only for pixels whose alpha value is greater than 0*. Optionally, merge with alpha information of a rectangle read through the **X** channel. The rectangle read through **Z** does not need to have the same size than the output rectangle.
Note: Executing this will modify the registers of the GFX MultiScaler.

Parameters: `[struct GFXEngine_MoveReplaceScaleRectangle_type]`

`[RMuint32] SrcX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **Z**.

`[RMuint32] SrcY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **Z**.

`[RMuint32] AlphaX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **X**.

`[RMuint32] AlphaY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **X**.

`[RMuint32] DstX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32] DstY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32] SrcWidth` Width of the rectangle read through **Z**.

`[RMuint32] SrcHeight` Height of the rectangle read through **Z**.

`[RMuint32] DstWidth` Width of the output rectangle (on surface **NX**). Also the width of the alpha input rectangle (read through surface **X**.

`[RMuint32] DstHeight` Height of the output rectangle (on surface **NX**). Also the height of the alpha input rectangle (read through surface **X**.

`[enum gfx_merge_mode] Merge`

GFX_MERGE_MODE_DISABLE The alpha information from the source (surface **Z**) is unmodified.

GFX_MERGE_MODE_X The **X** surface provides only the color information, and the alpha information is obtained from surface **X**.

GFX_MERGE_MODE_MODULATE The alpha information from channels **X** and **Z** is combined. *Note:* This mode is available only on SMP8630 chips.

Output: If the sizes of the source rectangle and the destination rectangle differ, the data read through **Z** is scaled to match the output size. Once that is done, every pixel on the output rectangle has one pixel from **Z** and one pixel from **X** associated, and the resulting output values are computed exactly as in the `MoveRectangle` command.

2.23 LinearGradientSurface

Setup a linear gradient. The gradient is between two 32bpp ARGB colors, and thus it is applied to the alpha component too.
Note: This command is available only on SMP8630 chips.

Parameters: `[struct GFXEngine_LinearGradientSurface_type]`

`[RMuint32] Width` Horizontal *period* of the gradient. Set to 0 for a vertical gradient.

`[RMuint32] Height` Vertical *period* of the gradient. Set to 0 for a horizontal gradient.

`[RMuint32] Color0` Upper-left color. In 32bpp ARGB format.

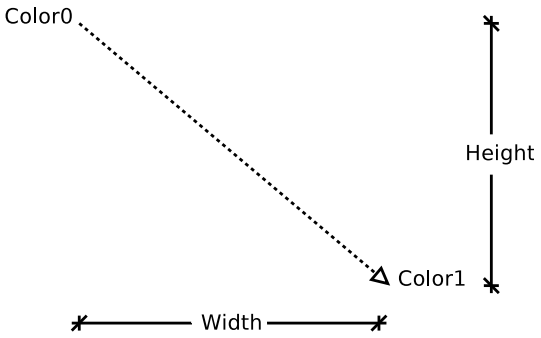


Figure 1: Linear Gradient Parameters. The dotted line represents the gradient direction.

`[RMuint32]` `Color1` Bottom-right color. In 32bpp ARGB format.

`[RMuint32]` `Weave` Four-bit field specifying on which components the engine should apply a weave effect.

- 0 means no weave.
- Add 1 for weave on the blue component.
- Add 2 for weave on the green component.
- Add 4 for weave on the red component.
- Add 8 for weave on the alpha component.

2.24 RadialGradientSurface

Setup a radial gradients. The gradient is between two 32bpp ARGB colors, and thus it is applied to the alpha component too. As shown on figure 2, two circles are defined, with one color assigned to each. The gradient applies only to the region between the two circles. The behaviors on the area inside the smaller circle and the area outside the bigger circle can be set up independently.
Note: This command is available only on SMP8630 chips.

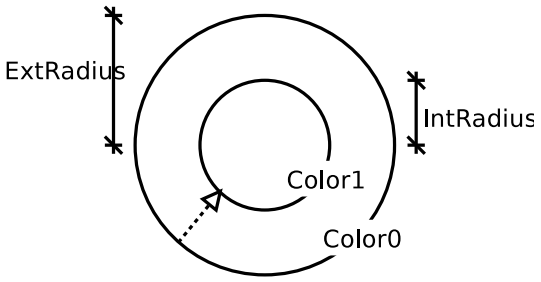


Figure 2: Radial Gradient Parameters. The dotted line represents the gradient direction.

Parameters: `[struct GFXEngine_RadialGradientSurface_type]`

`[RMuint32]` `IntRadius` Radius of the inner circle, in pixels. Can be 0.

`[RMuint32]` `ExtRadius` Redius of the external circle, in pixels. Must be bigger than `IntRadius`.

`[RMuint32]` `CenterX` Horizontal offset of the center of the circles, relative to the top of the drawing window. In pixels. Must be positive.

`[RMuint32]` `CenterY` Vertical offset of the center of the circles, relative to the left border of the drawing window. In pixels. Must be positive.

`[RMuint32]` `Color0` External color. In 32bpp ARGB format.

`[RMuint32]` `Color1` Internal color. In 32bpp ARGB format.

[RMbool] TransparentInt If set to TRUE, the are on the interior of the smaller circle is filled with transparent black (0x0). If set to FALSE, that area is filled with Color1.

[RMbool] TransparentExt If set to TRUE, the are on the exterior of the bigger circle is filled with transparent black (0x0). If set to FALSE, that area is filled with Color0.

[RMuint32] Weave Four-bit field specifying on which components the engine should apply a weave effect.

- 0 means no weave.
- Add 1 for weave on the blue component.
- Add 2 for weave on the green component.
- Add 4 for weave on the red component.
- Add 8 for weave on the alpha component.

2.25 BlendGradient

Does an alpha-blending of an internally-generated gradient and a rectangle read through **X**, and writes the resulting rectangle on the **NX** surface. The gradient provides the alpha information for the blend (see the Output description below). The parameters set by the last call to LinearGradientSurface or RadialGradientSurface will be used for gradient generation.
Note: This command is available only on SMP863X chips.

Parameters: [struct GFXEngine_BlendGradient_type]

[RMuint32] Src2X Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **X**.

[RMuint32] Src2Y Vertical offset of the top-left border of the rectangle, relative to the origin of surface **X**.

[RMuint32] DstX Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

[RMuint32] DstY Vertical offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

[RMuint32] Width Width of the blended rectangle.

[RMuint32] Height Height of the blended rectangle.

[RMbool] SaturateAlpha If set to TRUE, the alpha value of the resulting rectangle is set to 0xFF for all pixels.

Output: The value of a pixel on the output rectangle, Pix_{NX} is calculated from the two input pixels, Pix_X and Pix_G using the formulas described on the BlendRectangles command.

2.26 FillGradient

Fill a region of surface **NX** with an internally-generated gradient. Optionally, merge with alpha information of a rectangle read through the **X** channel. The parameters set by the last call to LinearGradientSurface or RadialGradientSurface will be used for gradient generation.
Note: This command is available only on SMP8630 chips.

Parameters: [struct GFXEngine_FillReplaceGradient_type]

[RMuint32] AlphaX Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **X**.

[RMuint32] AlphaY Vertical offset of the top-left border of the rectangle, relative to the origin of surface **X**.

[RMuint32] DstX Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

[RMuint32] DstY Vertical offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32]` `Width` Width of the filled rectangle.

`[RMuint32]` `Height` Height of the filled rectangle.

`[RMbool]` `Merge` If set to TRUE, the **X** surface is used to provide the alpha information. Otherwise the alpha information present on the gradient is used. See the `AlphaFormat` property for information on how to set up the **X** channel for this command.

Output: The value of a pixel on the output rectangle, Pix_{NX} is calculated from the two input pixels, $Alpha_X$ and the gradient (Pix_G) using the following formulas:

$$\begin{array}{ll} Alpha(Pix_{NX}) = Alpha_X & \text{If } \texttt{Merge} == \texttt{TRUE} \\ Alpha(Pix_{NX}) = Alpha(Pix_G) & \text{If } \texttt{Merge} == \texttt{FALSE} \end{array}$$

$$CB_i(Pix_{NX}) \quad = \quad CB_i(Pix_G)$$

Where CB_i represents each one of the three color components.

2.27 ReplaceGradient

Fill a region of surface **NX** with an internally-generated gradient, but *only for pixels whose alpha value is greater than 0*. Optionally, merge with alpha information of a rectangle read through the **X** channel. The parameters set by the last call to `LinearGradientSurface` or `RadialGradientSurface` will be used for gradient generation.
Note: This command is available only on SMP8630 chips.

Parameters: `[struct GFXEngine_FillReplaceGradient_type]`

`[RMuint32]` `AlphaX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **X**.

`[RMuint32]` `AlphaY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **X**.

`[RMuint32]` `DstX` Horizontal offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32]` `DstY` Vertical offset of the top-left border of the rectangle, relative to the origin of surface **NX**.

`[RMuint32]` `Width` Width of the filled rectangle.

`[RMuint32]` `Height` Height of the filled rectangle.

`[RMbool]` `Merge` If set to TRUE, the **X** surface is used to provide the alpha information. Otherwise the alpha information present on the gradient is used. See the `AlphaFormat` property for information on how to set up the **X** channel for this command.

Output: The value of a pixel on the output rectangle, Pix_{NX} is calculated from the two input pixels, $Alpha_X$ and the gradient (Pix_G) using the following formulas:

$$\begin{array}{ll} Alpha(Pix_{NX}) = Alpha_X & \text{If } \texttt{Merge} == \texttt{TRUE} \\ Alpha(Pix_{NX}) = Alpha(Pix_G) & \text{If } \texttt{Merge} == \texttt{FALSE} \end{array}$$

For every pixel, if $alpha\_value > 0$

$$\begin{aligned} Alpha(Pix_{NX}) &= alpha\_value \\ CB_i(Pix_{NX}) &= CB_i(Pix_G) \end{aligned}$$

Where CB_i represents each one of the three color components.
Otherwise (if $alpha\_value = 0$) the pixel on the **NX** surface is unmodified.

2.28 GlyphMask

Draw a glyph, without using the **NX** channel, at 1bpp.
The generated bitmap is written at the DRAM directly by the glyph engine (no need to assign a surface to channel **NX**), at the address specified by `OutAddr`. The glyph data is also directly read from DRAM, so no input channels need to be programmed either. The output format is always 1BPP, and the size is that of the scaled glyph's bounding box, with the width and height values incremented, if necessary, up to the first 64 pixels multiple (see below). It is up to the application to allocate memory space for the output, and make sure that it is big enough for the bitmap.
The maximum width of the generated bitmap is 4096 pixels.

Parameters: `[struct GFXEngine_GlyphMask_type]`

`[RMuint32] GlyphAddr` Start address the binary glyph (see Appendix C for details on its format). It is up to the application to allocate memory on DRAM and load the glyph before executing this command.

`[RMuint32] Size` Size of the glyph, in bytes. Glyphs should not exceed 1024 bytes.

`[RMuint32] OutAddr` Start address of the destination surface.

`[RMuint32] ScaleFactor` Ratio between the glyph metrics and the desired output size, with a $\frac{1}{2048}$ resolution:

$$\text{scale\_factor} = \left(\frac{\text{glyph grid units}}{\text{output size in pixels}} \right) * 2^{11}$$

The value is specified in 8.11 fixed point format, with unsigned integer part.

`[RMint16] XMax` Maximum absolute horizontal coordinate of the glyph (in glyph grid units).

`[RMint16] XMin` Minimum absolute horizontal coordinate of th glyph (in glyph grid units).

`[RMint16] YMax` Maximum absolute vertical coordinate of th glyph (in glyph grid units).

`[RMint16] YMin` Minimum absolute vertical coordinate of th glyph (in glyph grid units).

Output: The output is a 1BPP bitmap whose size in pixels can be calculated in the following manner:

```
/*  width of the scaled bounding box*/
width  = ((XMin-XMax)*ScaleFactor)>>11;
/* height of the scaled bounding box*/
height = ((YMin-YMax)*ScaleFactor)>>11;
/* make width and height 64-pixel multiples */
width  += ( width & 0x3F) ? (0x40 - ( width & 0x3F)) : 0);
height += (height & 0x3F) ? (0x40 - (height & 0x3F)) : 0);
```

2.29 GlyphScaleMatrix

Sets the parameters of an affine transformation. The transformation will be applied *only* to the next the `Glpyh` command that follow. This is typically used when drawing compound glyphs from truetype fonts.
Note: After a reset, the parameters are set to their default values (see below) so that no affine transformation is applied to the glyphs.

Parameters: `[struct GFXEngine_GlyphScaleMatrix_type]`

`[RMuint16] XScale` Specified in 2.14 fixed point format, with signed integer part. Default value (no transformation) is 1 (`0x4000h` in 2.14 format).

`[RMuint16] YScale`Specified in 2.14 fixed point format, with signed integer part. Default value (no transformation) is 1 (`0x4000h` in 2.14 format).

`[RMuint16] XYScale`Specified in 2.14 fixed point format, with signed integer part. Default value (no transformation) is 0 (`0x0h` in 2.14 format).

[RMuint16] YXScaleSpecified in 2.14 fixed point format, with signed integer part. Default value (no transformation) is 0 (0x0h in 2.14 format).

[RMint16] YOffset Specified as an integer, in glyph metric units. Default value (no transformation) is 0 (0x0h in 2.14 format).

[RMint16] XOffset Specified as an integer, in glyph metric units. Default value (no transformation) is 0 (0x0h in 2.14 format).

Transformation: The glyph points coordinates are transformed in the following way:

$$\begin{aligned} x' &= \texttt{XScale} * x + \texttt{XYScale} * y + \texttt{XOffset} \\ y' &= \texttt{YScale} * y + \texttt{YXScale} * x + \texttt{YOffset} \end{aligned}$$

Where x, y, are the original glyph point coordinates, and x', y' its transformed values.

2.30 FieldType

Sets the type of field of the surface connected to a channel. This is useful in cases where the GFXEngine is reading or writting from/to interlaced surfaces. If the field types of the input and output are correctly specified, the necessary filtering is done by the engine.

Parameters: [struct GFXEngine_FieldType_type]

[enum gfx_surface_id] SurfaceID Specifies to which channel the command should be applied.

[enum EMhwlibFieldType] FieldType Specifies the type of field that will be read/written through the channel. This value is only used for phase computation. Possible values are:

EMhwlibFieldType_Frame

EMhwlibFieldType_Top

EMhwlibFieldType_Bottom

[RMint32] LineSkipFactor Sets the line skip factor to be used when reading/writting from this surface. Setting this to N, forces that channel to skip (N-1) lines of the picture for every read/written line. For example, to read one field of an interlaced picture, this should be set to 2 (thus the other field will be skipped). Field selection is done by setting an appropriate vertical offset. Back to our example, to read the top field (even lines) of an interlaced picture, the vertical offset (Y) should be even, and to select a bottom field, it should be odd. If the value is negative, the image is read/written vertically reversed (bottomline first).

2.31 LPFThresholds

When an image is horizontally downscaled, chances are that some pixels need to be skipped. In those cases, applying an horizontal low-pass filter to the output often enhances the quality of the image. Typically, the stronger the pixel skip factor, the stronger should be the low-pass filtering. Since the amount of skipped pixels is not known by the application, this command sets the parameters that let the GFXEngine determine the intensity of the filtering in function of that factor.

Parameters: [struct GFXEngine_LPFThresholds_type]

[enum gfx_surface_id] SurfaceID Specifies to which channel the command should be applied.

[RMuint32] Threshold0 If $x\_downscale$ is below this threshold, and over Threshold1 and Threshold2 light filtering is applied. Default value is 384.

[RMuint32] Threshold1 If $x\_downscale$ is below this threshold, and over Threshold2 medium filtering is applied. Default value is 256.

[RMuint32] Threshold2 If $x\_downscale$ is below this threshold, strong filtering is applied. Default value is 128.

For all thresholds, the possible values range from 0 to 512. The $x\_downscale$ magnitude is defined as:

$$x\_downscale = \frac{512 * used\_pixels}{used\_pixels + skipped\_pixels}$$

For example, the case where no pixels need to be skipped corresponds to $x\_downscale = 512$, and the case where 3 pixels need to be skipped every 4 (very strong horizontal downscaling) corresponds to $x\_downscale = 128$.

2.32 BCS

The **Y** and **Z** channels, when scaling, can apply contrast, brightness and saturation transformations to the read image. This command allows to setup the paramaters. Note that by definition, this transformation makes only sense when applied to images in YUV colorspace. If applied to RGB surfaces, the brightness and contrast will affect the red plane, SaturationCb the green plane, and SaturationCr the blue plane. It is thus recommended to reset the paramaters to its neutral values before scaling RGB images.

Parameters: [struct GFXEngine_BCS_type]

[enum gfx_surface_id] SurfaceID Specifies to which channel the command should be applied.

[RMint32] Brightness Possible values range from -128 to 127. The default (neutral) value is 0.

[RMuint32] Contrast Possible values range from 0 to 255. The default (neutral) value is 128.

[RMuint32] SaturationCb Possible values range from 0 to 255. The default (neutral) value is 128.

[RMuint32] SaturationCr Possible values range from 0 to 255. The default (neutral) value is 128.

2.33 NonlinearScale

Parameters: [struct GFXEngine_NonlinearScale_type]

[enum gfx_surface_id] SurfaceID

[RMuint32] Level

[RMuint32] Width

[RMuint32] PARQuotient

A Multiple Buffering

A.1 Introduction to Multiple Buffering

The images on every monitor or TV are refreshed at a high frequency (up to 75Hz). This means that the displayed picture's pixels are read from DRAM as many times. Modifying this data while it is being displayed can be the cause of some undesirable artifacts, such as flickering or tearing. In most cases these effects are unacceptable, so applications should avoid to write on picture buffers that are being read by the display. Of course, most applications that make use of the GFXEngine need to write graphics into pictures and need to display those changes in real time. These applications need a way to guarantee that their graphic commands won't modify a picture buffer while it is being displayed.

In this section we introduce a method for doing so, called *multiple buffering*. It consists in creating multiple (typically two) picture buffers, instead of a single one. The changes are applied to the buffer that is not being displayed. Only when the changes are finished and the new picture needs to be displayed, the roles of the two buffers are swapped. This action is called *flipping*. Because flipping takes place during the vertical blanking interval, this method provides a way to display the changes without causing a tearing effect.

A.2 Multiple Buffering using the GFXEngine

As explained in the previous section, flipping the buffers can only be done during the blanking interval (i.e. once every time the output screen is refreshed). The GFXEngine API provides commands that force a synchronization between its graphic operations and the display events. This allows non-real-time applications to implement multiple buffering in a simple way. The commands that implement these features are `DisplayPicture` and `WaitForPicture`. The first one allows to schedule a given picture for its future display. The second one, allows to block the processing of all commands until a given picture is *no longer* being displayed. As we can see, both commands need to take some kind of picture buffer identifier as an argument. You can refer to the GFXEngine Properties section for information on the syntax.

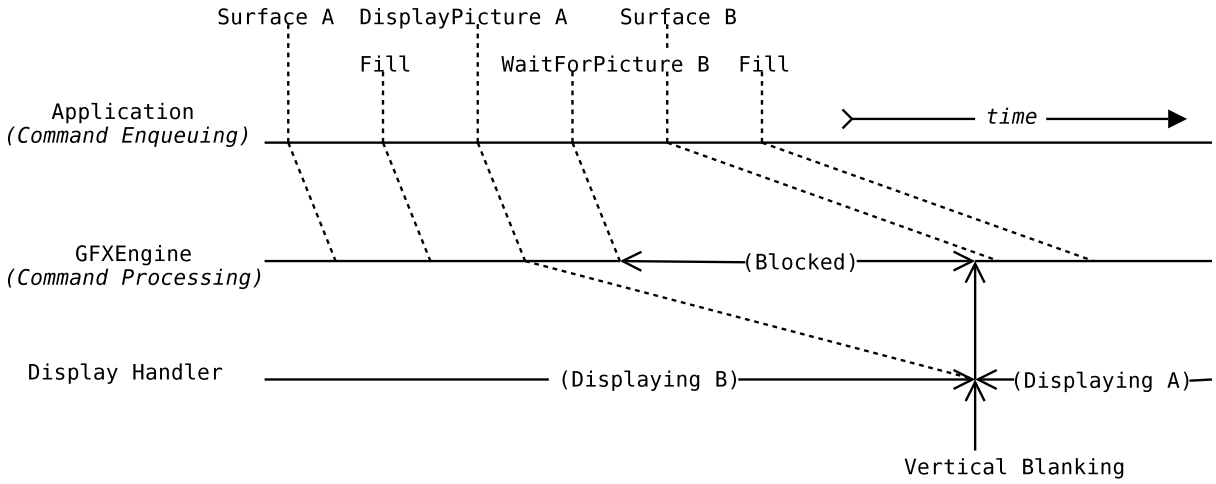


Figure 3: GFXEngine synchronization with the Display Handler

On the figure 3, A and B are two picture buffers. The top line represents the timing of the commands as they enter the GFXEngine command fifo. The second layer represents the timing of the commands as they are processed by the GFXEngine. Observe the DisplayPicture command is processed by the GFXEngine soon after it is enqueued, but it does not take effect (on the DisplayBlock) until the next vertical handling arrives. Also, note that once the WaitForPicture is processed, the GFXEngine stops until picture 'B' is no longer being displayed.

A typical application that uses the GFXEngine with multiple-buffering could be reduced to the following block of pseudo-code:

```
init_gfx_engine();

/* allocate the picture buffers and setup the needed surface structures */
setup_double_buffering();

while(!end){
   /* Obtain a new picture identifier */
   picture_id = get_next_picture_id();
   /* Make sure that a picture is not on display before writing on it */
   WaitForPicture picture_id;
   /* Setup the output of the GFXEngine to point to the new picture */
   Surface picture_id on NX;
   /* Perform various graphic operations on the new picture */
   draw_graphics();
   /* Schedule the finished picture for display */
   DisplayPicture picture_id;
}
```

A.3 Multiple Picture Surfaces

In order to implement multiple buffering, it is necessary that the picture buffers are part of a MultiplePictureSurface. This is a special type of surface provided by the EMhwlib. It contains a picture fifo, and optionally, an associated STC. Pictures that are enqueued on that fifo are scheduled for display. If a surface has an associated STC, a pts value must be assigned to every picture in the fifo. In that case, each picture is displayed when the STC reaches the pts value. Otherwise, the pictures in the fifo are displayed as soon as the next vertical blanking arrives. The DisplayBlock documentation contains detailed information how to create and setup MultiplePictureSurface's.

B Common Operations

B.1 Fading Transitions

For some applications it might be convenient to implement a smooth transition from one image into another, consisting in the first image vanishing while the other appears. In this section we describe two ways to achieve this 'fading' effect. One of them uses the GFXEngine, while the second only needs two scalers.

GFXEngine Fading

The fading effect can be easily achieved via the GFXEngine. To exemplify this we will imagine here and application fading from image A to image B. We need one (any) scaler to display the result, plus the GFXMultiScaler which will be used by the GFXEngine. To achieve an artifact-free result, double buffering is required. We will call the two display buffers X and Y.
There are many ways to implement this, but the one we present here has the advantage of not requiring both A and B to be present into DRAM at the same time. The idea is to initially display picture A (i.e. copy it into X and Y buffers) and then blend a nearly-transparent version picture B many times until only picture B is visible. This can be expressed in pseudo-code like this:

```
/* make image A available to the GFXEngine */
load_image(A);
Surface A on channel_Z
/* copy the first picture into the buffers */
flip_buffers(X, Y);
MoveAndScaleRectangles (A -> Y)
flip_buffers(Y, X);
MoveAndScaleRectangles (A -> X)
/* image A is no longer needed */
free_image(A);
/* make image B available to the GFXEngine */
load_image(B);
Surface B on channel_Z
/* setup alpha fading on Z */
AlphaPalette (set Alpha0, alpha1 to alpha_inc on Z)
EnableAlphaFading (enable on Z)
while(A_is_visible){
        pic1 = (pic1==X) ? Y:X;
        pic2 = (pic1==X) ? X:Y;
        flip_buffers(pic1, pic2); /*display pic1, writing on pic2 */
        /* put a some more B into the result */
        Surface pic1 on channel_X
        BlendAndScaleRectangles (B on pic1 -> pic2)
}
/* overwrite with B to make sure A is completely gone */
flip_buffers(X, Y);
MoveAndScaleRectangles (B -> Y)
flip_buffers(Y, X);
MoveAndScaleRectangles (B -> X)
/* image B is no longer needed */
free_image(B);
```

In the example, `flip_buffers(pic1, pic2)` handles double buffering as described in the "Multiple Buffering" section. Basically, `pic1` is displayed, the output channel is directed to pic2, and the `WaitForPicture` command is issued on pic2.
Now let's take a look at the line containing

`BlendAndScaleRectangles (B on pic1 -> pic2)` This means that B should be blended on top of pic1, and the result written on pic2. Note that this requires that the `X` and `Z` channels are set to pic1 and B, respectively. In the example, the `NX` channel is set to pic2 via the `flip_buffers()` function.
This command actually makes the destination buffer look a little bit more like picture B. Remember that the alpha used for a blend command is extracted from the top layer. In other words, the *amount* of image B that is blended on top of the existing buffer depends on image B's alpha plane. We need to control the alpha value of B's pixels to control the speed and smoothness of the fading.
A simple way to do that is to enable alpha fading on the channel that is to be used to read the image (`Z` in our example). Then Alpha0 and Alpha1 are set to some small value, via the `AlphaPalette` command. The resulting image, as seen by the GFXEngine, has all its pixels set to this constant value `alpha_inc`. This is what is done in the two lines preceding the loop. Of course, the smaller `alpha_inc`, the slower and smoother the fading will be. Note that if B's alpha plane is not constant, some fancy results can be obtained by setting different values for Alpha0 and Alpha1. The duration of the loop depends on `alpha_inc` and on the desired visual effect. It can be determined empirically. It's a good idea to end up copying B into X and Y to remove all the (small) remaining of A.

Scaler Fading

The simplest solution, which does not involve the use of the GFXEngine, consists on using two scalers to display the two images (or video streams, for that matter). The fading can be achieved by displaying both images at the same time and modifying their alpha plane to achieve the desired result.
For example, to fade from image A to image B, your application could display the first through the VCRMultiScaler, and the second one through the GFXMultiScaler. Both images are displayed at the same time, but the scalers are setup so that image B has an alpha value of 0x0, and image A has an alpha value of 0xff. Then B's alpha is progressively incremented and at the same time, A's alpha is decremented, until A is no longer visible.
We mentioned setting up a scaler to modify the image's alpha, but, how can this be done?
If the image doesn't have any alpha information, the scaler sets the desired alpha value as specified by its `Alpha0` property. So for the VCRMultiScaler (image A), you should gradually change the value its Alpha0 from 0xff to 0x0, and for the GFXMultiScaler, from 0x0 to 0xff.
If the image contains an alpha plane, modifying it can be achieved via the scaler's `EnableFading` property. In this case, a simple way is to enable the alpha fading and then proceed to change both Alpha0 and Alpha1 values like in the previous case.
If your application should fade off an image into a solid background, you only need to decrease the image's alpha value, and make sure that the mixer is in forced background mode. Please refer the mixer's `ForceBackGround` and `BackgroundColor` properties.

Note: All the scaler and mixer properties mentioned here are described in the display block documentation.

C Glyph Format

This section documents the binary format used by the GFX engine to describe glyphs.

C.1 What is a glyph?

A glyph is a vectorial representation of a binary bitmap, and its main advantage over a bitmap is that it is easily scalable. It consists on one or more closed paths, called contours. A contour is defined by means of a list of points. It is made of line segments (connecting two consecutive points), or Bézier arcs (whose shape is also defined by three or more consecutive points).
The inside of contours that are defined counter-clockwise is filled by the glyph engine, while contours defined clockwise determine white holes inside a filled area.
The points that define a contour are located on a grid of indivisible units. Possible values for the coordinates range from -16384 to 16383. The grid is oriented like the traditional mathematical two-dimensional plane, i.e., the X axis from the left to the right, and the Y axis from bottom to top.

C.2 The Glyph Binary Format

The data chunk pointed by the `GlyphAddr` field in the glyph-related commands, contains the glyph description in a binary format that is very similar to that used on the truetype font files.
Describing a glyph consists mainly in listing all the points that form the contours, and their properties.
As in the truetype format, the coordinates are specified incrementally (only the first point's position is specified with absolute glyph grid coordinates). The `RMGlyphPoints` can be seen as a vector that should be filled with the glyph data in the following way:

Value	Size in bytes
number of contours	2
zero padding	2
index of last point of first contour	2
...	
index of last point of n^{th} contour	2
...	
index of last point of last contour	2
zero padding until a 4-byte multiple	2 or none
flags for first point	1
x-coordinate for first point	1 or 2
y-coordinate for first point	1 or 2
...	
flags for n^{th} point	1 or none
x-coordinate for n^{th} point	1, 2 or none
y-coordinate for n^{th} point	1, 2 or none
...	
flags for last point	1 or none
x-coordinate for last point	1, 2 or none
y-coordinate for last point	1, 2 or none

For fields where there is a choice, the correct size is determined in function of the present or preceeding flags value (see the table below).
Note: For values that span over two bytes, the MSB should be written first.

Flag Name	Mask	Description
ON_CURVE	0x01	If set, the point is on the curve; otherwise it is off the curve (defines a Bézier arc)
X_SHORT	0x02	If set, the corresponding x-coordinate is 1-byte long. Otherwise, 2-bytes long.
Y_SHORT	0x04	If set, the corresponding x-coordinate is 1-byte long. Otherwise, 2-bytes long.
REPEAT	0x08	If set, the next byte specifies the number of additional times this set of flags is to be repeated.
SAME_X	0x10	This flag has two meanings, depending on how the X_SHORT flag is set. If X_SHORT is set, this bit describes the sign of the x-coordinate value, (1 means positive, 0 means negative). If X_SHORT is not set and this bit is et, then the current x-coordinate is the same as the previous x-coordinate. If the X_SHORT is not set and this bit is not set, the current x-coordinate is a singed 16-bit value.
SAME_Y	0x20	This flag has two meanings, depending on how the Y_SHORT flag is set. If Y_SHORT is set, this bit describes the sign of the y-coordinate value, (1 means positive, 0 means negative). If Y_SHORT is not set and this bit is et, then the current y-coordinate is the same as the previous y-coordinate. If the Y_SHORT is not set and this bit is not set, the current y-coordinate is a singed 16-bit value.
CUBIC	0x40	Set for cubic curves, unset for quadratic curves. Note that this bit is only valid if ON_CURVE is unset.