ORSoC Graphics accelerator Specification

Per Lenander, Anton Fosselius

March 23, 2012

Revision history

Rev.	Date	Author	Description
0.1.0	23/3/2012	Per Lenander	Initial draft

Contents

1 Introduction **5**
1.1 Features 5
1.2 IP Core directory structure 5

2 Architecture **5**
2.1 Overview 5
2.2 Concepts 7
2.3 Instruction fifo 8
2.4 Pipeline 8
2.5 Description of core modules 8
2.5.1 Wishbone slave 8
2.5.2 Vector processor 9
2.5.3 Rasterizer 9
2.5.4 Fragment processor 9
2.5.5 Blender 9
2.5.6 Wishbone arbiter 9
2.5.7 Wishbone master read 10
2.5.8 Renderer 10
2.5.9 Wishbone master write 10

3 IO Ports **10**

4 Registers **10**
4.1 Control Register (CTRL_REG) 11
4.2 Status Register (STATUS_REG) 11
4.3 Source Pixel position 0 Register (SRC_P0) 11
4.4 Source Pixel position 1 Register (SRC_P1) 11
4.5 Destination Pixel position 0 Register (DEST_P0) 11
4.6 Destination Pixel position 1 Register (DEST_P1) 12
4.7 Clip Pixel position 0 Register (CLIP_P0) 12
4.8 Clip Pixel position 1 Register (CLIP_P1) 12
4.9 Color Register (color) 12
4.10 Target addr Register (TADR_REG) 12
4.11 Target size Register (TSZE_REG) 13
4.12 Tex0 Base 13
4.13 Tex0 Size 13
4.14 Alpha 13
4.15 Colorkey 13

5 Operation **13**
5.1 Draw pixel 14
5.2 Fill rect 14
5.3 Line 14
5.4 Vector operations... 14

6 Clocks **14**

7 Driver interface **14**
7.1 newlib 14
7.1.1 oc_gfx_init 15
7.1.2 oc_vga_set_videomode 15
7.1.3 oc_vga_set_vbara 15
7.1.4 oc_vga_set_vbarb 15
7.1.5 oc_vga_bank_switch 15
7.1.6 oc_gfx_init_surface 15
7.1.7 oc_gfx_bind_rendertarget 16
7.1.8 oc_gfx_cliprect 16
7.1.9 oc_gfx_srcrect 16
7.1.10 oc_gfx_set_pixel 16
7.1.11 oc_gfx_memcpy 16
7.1.12 oc_gfx_set_color 16
7.1.13 oc_gfx_rect 17
7.1.14 oc_gfx_line 17
7.1.15 oc_gfx_enable_tex0 17
7.1.16 oc_gfx_bind_tex0 17
7.1.17 oc_gfx_enable_alpha 17
7.1.18 oc_gfx_set_alpha 17
7.1.19 oc_gfx_enable_colorkey 17
7.1.20 oc_gfx_set_colorkey 18
7.2 Extended newlib 18
7.2.1 oc_gfxplus_init 18
7.2.2 oc_gfxplus_init_surface 18
7.2.3 oc_gfxplus_bind_rendertarget 18
7.2.4 oc_gfxplus_flip 18
7.2.5 oc_gfxplus_clip 19
7.2.6 oc_gfxplus_fill 19
7.2.7 oc_gfxplus_line 19
7.2.8 oc_gfxplus_draw_surface 19
7.2.9 oc_gfxplus_draw_surface_section 19
7.2.10 oc_gfxplus_colorkey 20
7.2.11 oc_gfxplus_alpha 20
7.3 Linux 20
7.4 Utilities 20
7.4.1 Sprite Maker 20

8 Programming examples **20**

1 Introduction

The ORSoC Graphics accelerator allows the user to do advanced vector rendering and 2D blitting to a memory area. The core supports simple operations such as drawing textures, lines and filling rectangular areas with color.

This IP Core is designed to integrate with the OpenRISC processor through a Wishbone bus interface. The core itself has no means of displaying the information rendered, for this purpose it should work alongside a display component, such as the enhanced VGA/LCD IP core found on OpenCores.

1.1 Features

- 32-bit Wishbone bus interface
- Integrates with enhanced VGA/LCD IP core
- Support for 8, 16 and 32 bit color depth modes
- Support for variable resolution
- Acceleration of simple fill and line operations
- Acceleration of memory copy operations
- Textures can be saved to video memory
- Vector transformation and rasterization
- Clipping/Scissoring
- Alpha blending and colorkeying
- Requires 1800 Slice LUTs (Xilinx ISE 13.4)

1.2 IP Core directory structure

A basic overview of the contents of the IP core source folder can be found in figure 1. The **rtl** folder also contains files for implementing the component in ORPSoCv2.

2 Architecture

2.1 Overview

A basic topology of how the orgfx is connected to the VGA driver and OpenRisc core is shown in figure 2. The orgfx has three wishbone interfaces: one read-/write port that is used to communicate with the host CPU. One read port that reads texture/alpha blending information from the RAM and one write port to write pixel information to the RAM.

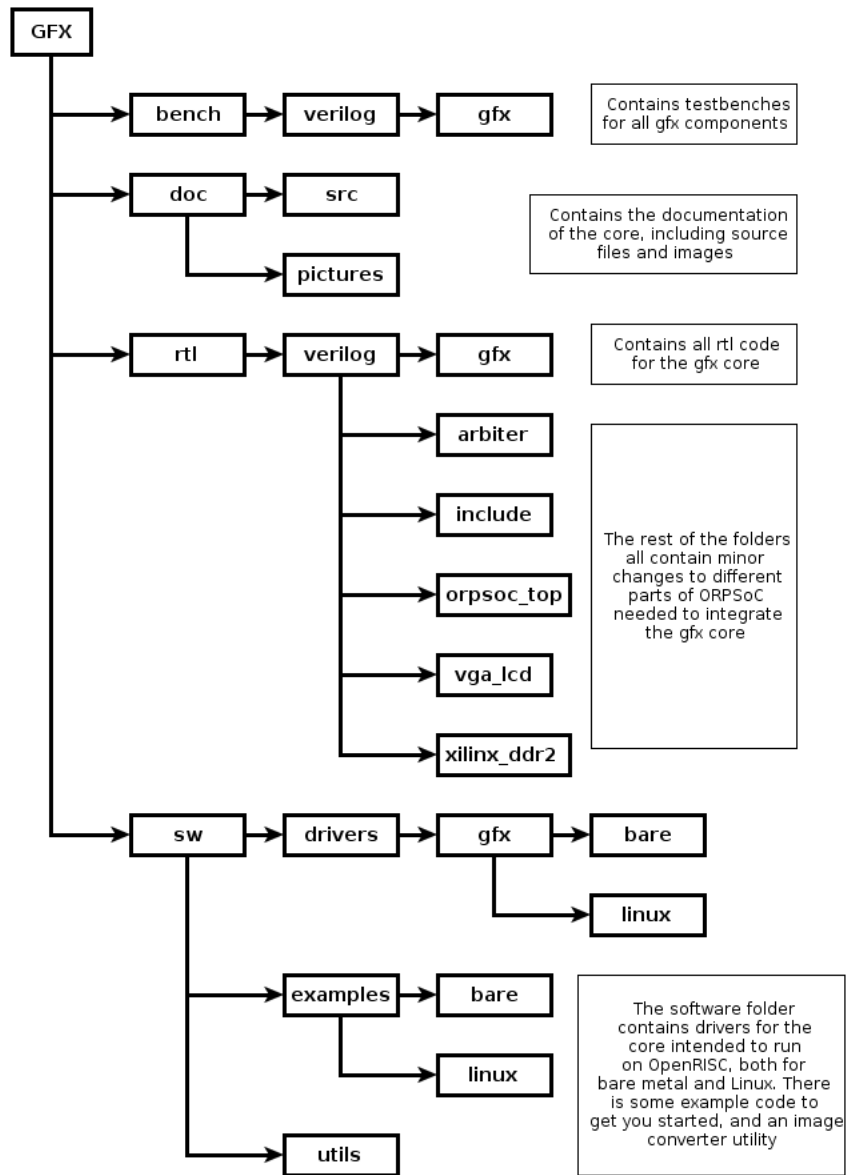

Figure 1: Directory structure of the ORSoC graphics accelerator.

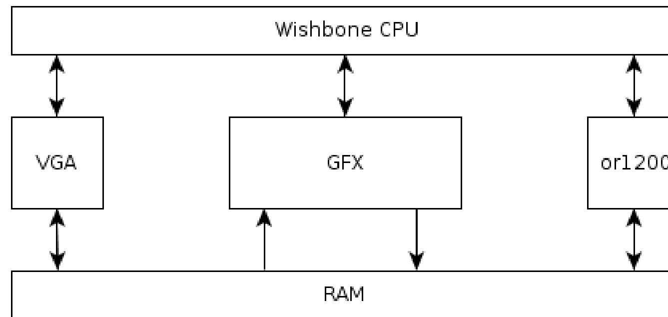

Figure 2: Overview of the ORPSoCv2's wishbone interconnection.

2.2 Concepts

This section describes a few basic terms used in this document.

Video memory – The orgfx component writes pixels one by one to an external memory, usually a SDRAM or DDR RAM chip. The CPU should also have access to this memory space to be able to write pixels directly (the easiest way to load textures).

Render target – The render target, defined by the target base and size registers, describes the area to which all operations render pixels. It is possible to change the base address and size, enabling render-to-texture and double buffering.

Surface/Texture – Any memory area that can be rendered to, including the render target, is considered a surface. A surface is defined by it's base address and size. There are two main surfaces that the orgfx device handles: the render target and the currently active texture. Swapping between different textures has to be done in software. The operation of setting the current render target or texture is referred to as *binding.*

Source, Destination and Clip rectangles – There are three sets of rectangles that affect rendering, each described by two points. The first point sets the beginning of the rectangle, while the second point sets the pixel after the end of the rectangle. This way, a rectangle exactly filling the screen would be (0,0,640,480) at 640x480 resolution. See figure 3;

Source rectangle – The source rectangle defines what pixels should be read from a texture during textured operations. The points are defined in the coordinates of the currently bound texture. This way sections of a texture can be drawn (good for tile maps or bitmap fonts).

Destination rectangle – The destination rectangle defines where operations such as draw pixel and draw line will draw pixels, in the coordinates of the render target.

Clip rectangle – The clip rectangle defines an area within the current render target which is valid to draw to. Any pixels outside this rectangle are discarded in the rasterization step. Pixels outside of the render target are automatically discarded.

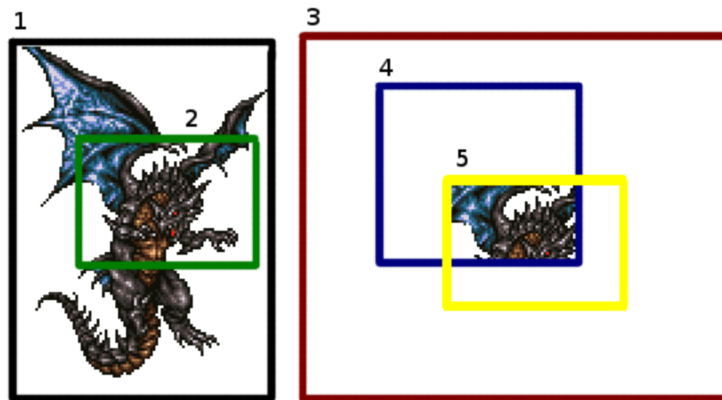

Figure 3: 1. Texture, 2. Source, 3. Render target, 4. Clip, 5. Destination

2.3 Instruction fifo

All wishbone writes sent to the slave interface will pass through an instruction fifo. If the device is in the busy state (when the pipeline is active) the instruction will be queued instead of being set immediately. This is important to take into account when reading from registers, since an operation that changes the register being read might be queued. To find out if the device is busy, poll the status register and check if the busy bit is high.

2.4 Pipeline

Currently the pipeline begins with the **rasterizer**, its purpose is to convert points into pixels. When there are a request to write a rectangle, the **rasterizer** receives two points and outputs all the pixels between those points. The pixels generated by the **rasterizer** in recieved by the **fragment processor**. In the **fragment processor** the pixel gets a color either by a pre-set color or from a texture (via the **wishbone read** from the **memory**). The colored pixel is then sent to the **blender** which handles alpha blending. If alpha is enabled, the **blender** will read from the address in the **memory** where the pixel is going to be drawn. Then the new color is calculated based on the pixel color and the alpha value. The pixel is then passed on to the **renderer** which calculates the address in the memory where the pixel is going to be written. The pixel is then passed on to the **wishbone write** interface and finally to the render target in the **memory**.

2.5 Description of core modules

2.5.1 Wishbone slave

The wishbone slave handles all communication from the main OpenRISC processor (or other master cpu). This component holds all the registers, and the instruction fifo that sets them. This component can be in one of two states:

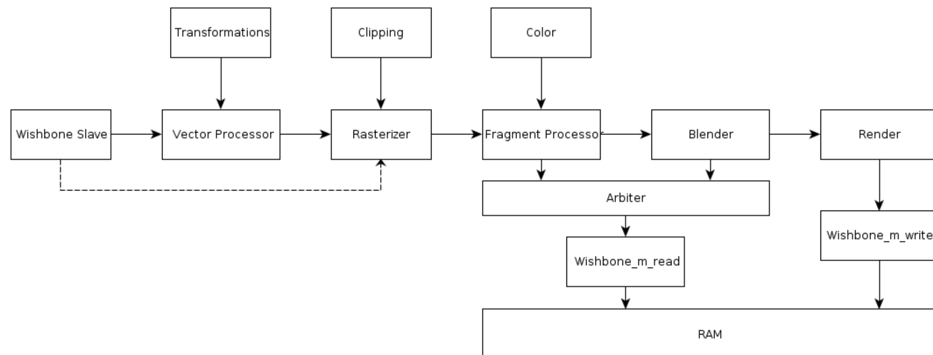

Figure 4: Picture of the orgfx pipeline

busy or *wait*. It enters the busy state when a pipeline operation is initialized, and returns to the wait state when the operation is finished.

2.5.2 Vector processor

This module is a stub for future releases.

2.5.3 Rasterizer

This module generates pixel coordinates for several different operations.

2.5.4 Fragment processor

The fragment processor adds color to the pixel generated by the rasterizer. If texturing is disabled, a color supplied from the color register is used. If texturing is enabled on the other hand, the u v coordinates supplied by the rasterizer are used to fetch a pixel from the active texture. If colorkeying is enabled and the fetched color matches the color key, the current pixel is discarded.

2.5.5 Blender

The blender module performs alpha blending if this is enabled. The module fetches the color of the pixel that the current operation will write to, and the mixes the value of the target color and the color from the fragment processor using the following algorithm:

$$color_{out} = color_{in} * alpha + color_{target} * (1 - alpha)$$

Where alpha is a value between 0 (transparent) and 1 (opaque). If alpha blending is disabled the pixel is passed on unmodified.

2.5.6 Wishbone arbiter

Since two parts of the pipeline (fragment and blender) needs to access video memory, the arbiter makes certain only one of them can access the reader at once. The blender has the highest priority.

2.5.7 Wishbone master read

The wishbone reader handles all reads from video memory.

2.5.8 Renderer

The renderer calculates the memory address of the target pixel.

2.5.9 Wishbone master write

The wishbone master handles all writes to the video memory.

3 IO Ports

The Core has three wishbone interfaces:

- Wishbone slave – connects to the data bus of the OpenRISC processor. In the case of ORPSoC, this bus is connected through an arbiter.
- Wishbone master read-only – connects to a video memory port with read access. Used for fetching textures and during blending.
- Wishbone master write-only – connects to a video memory port with write access. Used for rendering pixels to the framebuffer.

There is an interrupt enabled that can be connected to the interrupt pins on the or1200 CPU (in the supplied orpsoc_top it is connected to or1200_pic_ints[9]). For this interrupt to trigger, the correct bits in the control register has to be set.

4 Registers

Name	Addr	Width	Access	Description
Control	0x00	32	RW	Control register
Status	0x04	32	R	Status register
Src Pixel0	0x08	32	RW	Source pixel0 register
Src Pixel1	0x0c	32	RW	Source pixel1 register
Dest Pixel0	0x10	32	RW	Destination pixel0 register
Dest Pixel1	0x14	32	RW	Destination pixel1 register
Clip Pixel0	0x18	32	RW	Clip pixel0 register
Clip Pixel1	0x1c	32	RW	Clip pixel1 register
Color	0x20	32	RW	Color register
Target Base	0x24	32	RW	Render target base address
Target Size	0x28	32	RW	Render target width/height register
Tex0 Base	0x2C	32	RW	Texture0 base
Tex0 Size	0x30	32	RW	Texture0 size
Alpha	0x34	32	RW	Global alpha register
Colorkey	0x38	32	RW	Colorkey register

Each register is described in detail in the following sections, with information about what the purpose of each bit in the register is. The default value provided for each register is set when the device receives a reset signal.

4.1 Control Register (CTRL_REG)

Bit #	Access	Description
[31:10]	-	Reserved
[9]	RW	Line write
[8]	RW	Rect write
[7:5]	-	Reserved
[4]	RW	Colorkey enable
[3]	RW	Blending enable
[2]	RW	Texture0 enable
[1:0]	RW	Color depth

Default value: 0x00

Color depth is defined as follows:

Mode	Color depth
00	8 bit
01	16 bit
10	24 bit (not supported)
11	32 bit

4.2 Status Register (STATUS_REG)

Bit #	Access	Description
[31:1]	R	Reserved
[0]	R	Busy pin (high when busy)

Default value: 0x00

4.3 Source Pixel position 0 Register (SRC_P0)

Bit #	Access	Description
[31:16]	RW	x
[15:0]	RW	y

Default value: 0x00

The source pixels are used to define a specific area in a texture to draw.

4.4 Source Pixel position 1 Register (SRC_P1)

Bit #	Access	Description
[31:16]	RW	x
[15:0]	RW	y

Default value: 0x00

4.5 Destination Pixel position 0 Register (DEST_P0)

Bit #	Access	Description
[31:16]	RW	x
[15:0]	RW	y

Default value: 0x00

4.6 Destination Pixel position 1 Register (DEST_P1)

Bit #	Access	Description
[31:16]	RW	x
[15:0]	RW	y

Default value: 0x00

4.7 Clip Pixel position 0 Register (CLIP_P0)

Bit #	Access	Description
[31:16]	RW	x
[15:0]	RW	y

Default value: 0x00

4.8 Clip Pixel position 1 Register (CLIP_P1)

Bit #	Access	Description
[31:16]	RW	x
[15:0]	RW	y

Default value: 0x00

4.9 Color Register (color)

Bit #	Access	Description
[31:0]	RW	Color bits

Default value: 0x00

There are several color modes available (set in *video mode register*):

Mode	Format
32bpp	[31:24] is alpha channel. [23:16] is R, [15:8] is G and [7:0] is B
16bpp	[15:11] is R, [10:5] is B and [4:0] is G
8bpp gray	[7:0] sets both R, G and B values
8bpp palette	[7:0] sets the color index in the palette

4.10 Target addr Register (TADR_REG)

Bit #	Access	Description
[31:2]	RW	Video Memory Address
[1:0]	-	Nothing

Default value: 0x00

4.11 Target size Register (TSZE_REG)

Bit #	Access	Description
[31:16]	RW	Width
[15:0]	RW	Height

Default value: 0x00

4.12 Tex0 Base

Bit #	Access	Description
[31:2]	RW	Video Memory Address
[1:0]	-	Nothing

Default value: 0x00

4.13 Tex0 Size

Bit #	Access	Description
[31:16]	RW	Width
[15:0]	RW	Height

Default value: 0x00

4.14 Alpha

Bit #	Access	Description
[31:8]	-	Reserved
[7:0]	RW	Global alpha

Default value: 0xff

The global alpha value is used in all rendering when alpha blending is enabled. **0xff** is full opacity, while **0x00** is full transparency (nothing rendered).

4.15 Colorkey

Bit #	Access	Description
[31:0]	RW	Colorkey

Default value: 0x00

By setting a colorkey certain pixels in a texture can be discarded in the fragment stage, providing a hard transparency. Depending on the color depth, a mask is applied to the color. Using 8 bit color, only the 8 least significant bits in the colorkey will be compared with the texture color during the check. The colorkey enable bit in the control register must be set to enable this functionality.

5 Operation

All hardware accelerated operations draw pixels to the currently active surface (defined by TADR_REG and TSZE_REG). These operations are all affected by clip_p0 and clip_p1. No pixels that fall outside the clipping rectangle will be rasterized.

5.1 Draw pixel

Input needed: dest_p0, color
Orgfx have no hardware-support for writing a single pixel to the video memory. However the software API makes this operation possible by writing directly to the memory. Since the video memory can point to both the framebuffer and to textures, the same operation can be used to draw an arbitrary pixel to the screen and to load a texture into video memory.

5.2 Fill rect

Input needed: ctrl, dest_p0, dest_p1, color, [src_p0, src_p1]
Fill rect will fill the area of a rectangle created between the pixel dest_p0 and dest_p1 with color. If texturing is enabled, color will be taken from the active texture in the area between src_p0 and src_p1. This operation is hardware accelerated, and is activated by setting the Rect write bit in the control register.

5.3 Line

Input needed: dest_p0, dest_p1, color
Line will draw a line between the pixels dest_p0 and dest_p1 with color. This operation is hardware accelerated.

5.4 Vector operations...

6 Clocks

The wishbone slave uses the system wishbone bus clock at 50 Mhz, while the rest of the pipeline and the wishbone interfaces to the memory runs at 100Mhz.

7 Driver interface

The ORSoC graphics accelerator offers three different APIs to code against, two for bare metal when coding directly against the processor, and a Linux kernel module. The extended bare metal interface is a wrapper around the basic bare metal API, and makes coding easier by reducing the number of calls. The drawback is lesser control over the graphics card.

7.1 newlib

The basic library is provided in **oc_gfx.h** and **oc_gfx.c**.

The bare metal library declares a structure that can hold surfaces (both framebuffers and textures). Many functions take a pointer to one of these structures.

```
struct oc_gfx_surface
{
        unsigned int addr;
        unsigned int w;
        unsigned int h;
```

```
};
```

7.1.1 oc_gfx_init

Description: The oc_gfx_init must be called first to get other oc_gfx commands to work properly.

```
void oc_gfx_init(unsigned int memoryArea);
```

7.1.2 oc_vga_set_videomode

Description: Sets the video mode, width, height, bpp.

```
void oc_gfx_set_videomode(unsigned int width,
                          unsigned int height,
                          unsigned char bpp);
```

7.1.3 oc_vga_set_vbara

Description: Assign a memory address to "Video Base Address Register A".

```
void oc_vga_set_vbara(unsigned int addr);
```

7.1.4 oc_vga_set_vbarb

Description: Assign a memory address to "Video Base Address Register B".

```
void oc_vga_set_vbarb(unsigned int addr);
```

7.1.5 oc_vga_bank_switch

Description: Switches the framebuffer.

```
void oc_vga_bank_switch();
```

7.1.6 oc_gfx_init_surface

Description: Initialize a surface and return a control structure for it. This function increments an internal video memory stack pointer, so each surface will be allocated after the previous one in memory (starting at memoryArea set by oc_gfx_init). There is currently no memory management in place to recycle surface memory once it is no longer in use. The first surface initialized will point to the same memory that the video controller reads from, so it should be initialized with the width and height of the screen.

```
struct oc_gfx_surface
  oc_gfx_init_surface(unsigned int width,
                      unsigned int height);
```

7.1.7 oc_gfx_bind_rendertarget

Description: Binds a surface as the active render target. This function *must* be called before any drawing operations can be performed.

```
void oc_gfx_bind_rendertarget(struct oc_gfx_surface *surface);
```

7.1.8 oc_gfx_cliprect

Description: Sets the clipping rect. No pixels will be drawn outside of this rect (useful for restricting draws to a specific area of the render target). oc_gfx_bind_rendertarget will reset the clipping rect to the size of the surface.

```
inline void oc_gfx_cliprect(unsigned int x0,
                            unsigned int y0,
                            unsigned int x1,
                            unsigned int y1);
```

7.1.9 oc_gfx_srcrect

Description: Sets the source rectangle that will be used by texturing operations. This allows for only drawing a small part of a texture. oc_gfx_bind_tex0 will reset this to the size of the texture.

```
inline void oc_gfx_srcrect(unsigned int x0,
                           unsigned int y0,
                           unsigned int x1,
                           unsigned int y1);
```

7.1.10 oc_gfx_set_pixel

Description: Set a pixel on coordinate x,y to color. This is done in software by direct memory writes. This operation is not affected by the clipping rect!

```
inline void oc_gfx_set_pixel(unsigned int x,
                             unsigned int y,
                             unsigned int color);
```

7.1.11 oc_gfx_memcpy

Description: Copies memory from the processor to the video memory. Size is in 32-bit words. This function is intended to work with the output array of the sprite converter utility to load images into memory. Remember to bind a texture as the render target first!

```
void oc_gfx_memcpy(unsigned int mem[],
                   unsigned int size);
```

7.1.12 oc_gfx_set_color

Description: Sets the current drawing color.

```
inline void oc_gfx_set_color(unsigned int color);
```

7.1.13 oc_gfx_rect

Description: Draws a rect from x0,y0 to x1,y1 and fills it with the current drawing color. If texturing is enabled, the current texture will be drawn instead.

```
inline void oc_gfx_rect(unsigned int x0,
                        unsigned int y0,
                        unsigned int x1,
                        unsigned int y1);
```

7.1.14 oc_gfx_line

Description: Draws a line from x0,y0 to x1,y1 with the current drawing color. If texturing is enabled, the first pixel of the current texture will be drawn instead.

```
inline void oc_gfx_line(unsigned int x0,
                        unsigned int y0,
                        unsigned int x1,
                        unsigned int y1);
```

7.1.15 oc_gfx_enable_tex0

Description: Enables or disables texturing.

```
void oc_gfx_enable_tex0(unsigned int enable);
```

7.1.16 oc_gfx_bind_tex0

Description: Binds a surface as the current texture. Will reset the source rect.

```
void oc_gfx_bind_tex0(struct oc_gfx_surface* surface);
```

7.1.17 oc_gfx_enable_alpha

Description: Enables or disables alpha blending.

```
void oc_gfx_enable_alpha(unsigned int enable);
```

7.1.18 oc_gfx_set_alpha

Description: Sets the alpha blending value.

```
void oc_gfx_set_alpha(unsigned int alpha);
```

7.1.19 oc_gfx_enable_colorkey

Description: Enables or disables colorkey.

```
void oc_gfx_enable_colorkey(unsigned int enable);
```

7.1.20 oc_gfx_set_colorkey

Description: Sets the colorkey color.

```
void oc_gfx_set_colorkey(unsigned int colorkey);
```

7.2 Extended newlib

The extended library is provided in **oc_gfx_plus.h** and **oc_gfx_plus.c**, but **oc_gfx.c** also has to be compiled for it to work.

Instead of using surface structs directly, the extended API hides surface management by returning id tags for each surface. The screen surface (defined by id -1) is handled as a single surface, even when double buffering is enabled.

The driver defines the number of available surfaces (not counting the screen) with a static define. Change this if the default value is too low for your application.

7.2.1 oc_gfxplus_init

Description: Initializes the screen with the supplied video mode and returns an id for the screen.

```
int oc_gfxplus_init(unsigned int width,
                    unsigned int height,
                    unsigned char bpp,
                    unsigned char doubleBuffering);
```

7.2.2 oc_gfxplus_init_surface

Description: Unlike the basic API, this function both initializes a surface and loads a prepared image to it in one function call. The return value is an id that can be used to bind the surface. It changes render target during operation, but switches back to the last render target on completion. Since the screen(s) are already initialized by a call to init, they do not need to be loaded using this function.

```
int oc_gfxplus_init_surface(unsigned int width,
                            unsigned int height,
                            unsigned int mem[]);
```

7.2.3 oc_gfxplus_bind_rendertarget

Description: Binds a surface as the current render target.

```
void oc_gfxplus_bind_rendertarget(int surface);
```

7.2.4 oc_gfxplus_flip

Description: Swaps which buffer to draw on when using double buffering. Needs to be called once before anything shows up on screen!

```
void oc_gfxplus_flip();
```

7.2.5 oc_gfxplus_clip

Description: Sets the current clipping rect. This is reset to the size of the new render target when oc_gfxplus_bind_rendertarget is called.

```
inline void oc_gfxplus_clip(unsigned int x0,
                            unsigned int y0,
                            unsigned int x1,
                            unsigned int y1);
```

7.2.6 oc_gfxplus_fill

Description: Draws a rectangle to the current render target with a flat color.

```
void oc_gfxplus_fill(unsigned int x0,
                     unsigned int y0,
                     unsigned int x1,
                     unsigned int y1,
                     unsigned int color);
```

7.2.7 oc_gfxplus_line

Description: Draws a line from x0,y0 to x1,y1 to the current render target with a flat color.

```
void oc_gfxplus_line(unsigned int x0,
                     unsigned int y0,
                     unsigned int x1,
                     unsigned int y1,
                     unsigned int color);
```

7.2.8 oc_gfxplus_draw_surface

Description: Draws a texture to the current render target.

```
void oc_gfxplus_draw_surface(unsigned int x0,
                             unsigned int y0,
                             unsigned int surface);
```

7.2.9 oc_gfxplus_draw_surface_section

Description: Draws a section of a texture defined by src0, src1 to the current render target.

```
void oc_gfxplus_draw_surface_section(unsigned int x0,
                                     unsigned int y0,
                                     unsigned int srcx0,
                                     unsigned int srcy0,
                                     unsigned int srcx1,
                                     unsigned int srcy1,
                                     unsigned int surface);
```

7.2.10 oc_gfxplus_colorkey

Description: Sets the colorkey color and enables or disables the use of the colorkey.

```
void oc_gfxplus_colorkey(unsigned int colorkey,
                         unsigned int enable);
```

7.2.11 oc_gfxplus_alpha

Description: Sets the alpha value and enables or disables the use of the alpha blending.

```
void oc_gfxplus_alpha(unsigned int alpha,
                      unsigned int enable);
```

7.3 Linux

The current version of the core does not have a Linux driver.

7.4 Utilities

7.4.1 Sprite Maker

Since there is no libraries for loading images in the bare metal driver, a utility program is provided that converts an image into a format that can be loaded to the graphics accelerator. The Sprite Maker utility uses SDL and SDL_image to load images, and supports loading several basic formats, such as bmp, jpg, png, gif etc. The utility supports writing to 8-, 16-, 24- and 32-bits-per-pixel (must match the format set by oc_gfx_set_videomode). The width of the loaded image must be a multiple of 4 pixels (8 bpp), 2 pixels (16 bpp) or 1 bpp (24, 32 bpp) respectively.

The resulting output of the utility is a header file that can be included into your program. This header declares an array, which can be copied to memory and be used as a texture.

This is sample shows how the converter utility can be used:

```
./spritemaker image.png [bpp]
```

If bpp is not provided, the utility uses 8 bits-per-pixel. For an example of how to use the output of the converter, see section 8.

8 Programming examples

The following piece of code shows how to use the extended interface for a bare metal implementation on the ORPSoCv2 platform. Bahamut_cc.png.h is a 186 by 248 pixel image with a pinkish background (rgb code ff00ff, or f81f in 16 bit). The header file is generated by the sprite maker utility at 16 bit color depth.

```
#include "oc_gfx_plus.h"

#include "Bahamut_cc.png.h"
```

```
int main(void)
{
    int i;

    // Initialize screen to 640x480-16@60
    // No double buffering
    int screen = oc_gfxplus_init(640, 480, 16, 0);

    // Initialize dragon sprite
    int bahamut_sprite =
        oc_gfxplus_init_surface(186, 248, Bahamut_cc);

    // Activate colorkeying
    oc_gfxplus_colorkey(0xf81f, 1);

    // Clear screen, white color
    oc_gfxplus_fill(0,0,640,480,0xffff);
    // Draw a few lines with different colors
    oc_gfxplus_line(200,100,10,10,0xf000);
    oc_gfxplus_line(200,100,351,31,0x0ff0);
    oc_gfxplus_line(200,100,121,231,0x00f0);
    oc_gfxplus_line(200,100,321,231,0xf00f);

    // Draw the dragon at different alpha settings
    oc_gfxplus_alpha(64,1);
    oc_gfxplus_draw_surface(100, 100, bahamut_sprite);
    oc_gfxplus_alpha(128,1);
    oc_gfxplus_draw_surface(120, 102, bahamut_sprite);
    oc_gfxplus_alpha(255,1);
    oc_gfxplus_draw_surface(140, 104, bahamut_sprite);

    while(1);
}
```

References