1 File *kgi.c*

The file *kgi.c* contains the heart of the KGI system. It manages all displays available and all devices using these displays.

1.1 Terminology

In order to understand KGI it is essential to have clear understanding of KGI terminology.

display is a physical display device such as a graphic card. Displays get installed only at boot time. Due to the way different maps are set up, it is impossible to add or remove a display dynamically. However, it is possible to *replace* a display.

device is a particular view of a display. For instance, every virtual terminal is a device. Every device needs to *register* itself with the display. In order for the device to take active posession of the display, the device needs to be *mapped* to the display. Each device maintains its display mode which gets set when the device is mapped.

console is essentially a device. The only difference is that a userland application cannot create cosoles, they are completely separated from (so called) graphic devices created by processes.

focus is a collection of input devices.

1.2 KGI maps

The display configuration is maintained in a couple of global maps.

```
kgi_u8_t console_map[CONFIG_KGII_MAX_NR_FOCUSES][CONFIG_KGII_MAX_NR_CONSOLES];
kgi_u8_t graphic_map[CONFIG_KGII_MAX_NR_FOCUSES][CONFIG_KGII_MAX_NR_CONSOLES];
kgi_u8_t focus_map[CONFIG_KGII_MAX_NR_DEVICES];
kgi_u8_t display_map[CONFIG_KGII_MAX_NR_DEVICES];

static kgi_device_t     *kgidevice[KGI_MAX_NR_DEVICES];
static kgi_display_t    *kgidpy[KGI_MAX_NR_DISPLAYS];
```

console_map maps focus number and console id to a device id

graphic_map maps fucus number and graphic device id to device id

focus_map maps device ids to focus ids

display_map maps device ids to display ids

kgidevice maps device ids to their *kgi_device_t* structures

kgidpy maps display ids to their *kgi_display_t* structures

Probably the best way to understand the role these maps play is to look at how KGI initializes them.

```
static inline void kgi_init_maps(kgi_u_t nr_displays, kgi_u_t nr_focuses)
{
        kgi_u_t display = 0, device;
```

Currently KGI is configured to use 128 devices. The first 64 are console devices and the second 64 are so called graphic devices. The following loop will iterate over each console and graphic device

```
        for (device = 0; device < CONFIG_KGII_MAX_NR_CONSOLES; device++) {
```

Each focus is capable of registering 16 consoles and 16 graphic devices, so with 64 consoles and 64 graphic devices there can be 4 focuses

```
                kgi_u_t focus   = device / 16;
                kgi_u_t console = device % 16;
```

Sanity check (mostly making sure the configuration values are valid)

```
                if (! (KII_VALID_FOCUS_ID(focus) &&
                        KII_VALID_CONSOLE_ID(console))) {

                        continue;
                }

                KRN_DEBUG(3, "mapping device %i on focus %i, display %i",
                        device, focus, display);
```

console_map and *graphic_map* specify the device given a focus and the device number. Lower half of all devices is used for consoles and upper half is used for graphic devices, so the *console_map* is initialized with the device number and *graphic_map* is initialized with the device number plus half the total number of devices.

```
                console_map[focus][console] = device;
                graphic_map[focus][console] = device +
                        CONFIG_KGII_MAX_NR_CONSOLES;
```

focus_map and *display_map* are the inverse mappings, mapping devices to focus and display

```
		focus_map[device] =
		focus_map[device + CONFIG_KGII_MAX_NR_CONSOLES] = focus;

		display_map[device] =
		display_map[device + CONFIG_KGII_MAX_NR_CONSOLES] = display;
```

In order to create a sensible configuration for any number of focuses and displays, some heuristics are applied. Each display is used by 16 consoles and 16 graphic devices so when *console* gets to 15 the display number is incremented. However, in the case where there are more displays than focuses on the system each display is mapped to only 4 consoles and 4 graphic devices. This effectively allows the extra displays to be controlled by a single focus.

```
		if ((device % 4) == 3) {

			if (nr_displays > nr_focuses) {

				display++;
				nr_displays--;
			}
		}
		if (console == 15) {

			nr_focuses--;
			nr_displays--;
			display++;
		}
	}
}
```

So, given the default configuration the setup will be as follows. There are in total 128 devices. The first 64 of these are console devices which are mapped to focuses and displays in groups of 16 (in case there are less displays than focuses they will be mapped to display 4 at a time). The remaining 64 devices are graphic devices and are mapped to focuses and displays in the same manner.

1.3 Boot time initialization

On boot *kgi_init* initializes all maps and creates all available displays.

```
void kgi_init(void)
{
	kgi_u_t nr_displays, nr_focuses;
```

Clear all maps to invalid values

```
        memset(display_map, 0xFF, sizeof(display_map));
        memset(focus_map,   0xFF, sizeof(focus_map));
        memset(console_map, 0xFF, sizeof(console_map));
        memset(graphic_map, 0xFF, sizeof(graphic_map));

        memset(kgidpy, 0, sizeof(kgidpy));
        memset(kgidevice, 0, sizeof(kgidevice));
```

Here all displays ever present on the system are initialized. KGI is unable to add displays, it can only *replace* them. Thus a couple of "dummy" null displays are created. These displays exist soley to be replaced by other (most likey native) displays.

```
        nr_displays = 0;
#ifdef  CONFIG_KGI_DPY_I386
        nr_displays = dpy_i386_init(nr_displays, CONFIG_KGI_DISPLAYS);
#endif
#ifdef  CONFIG_KGI_DPY_NULL
        nr_displays = dpy_null_init(nr_displays, CONFIG_KGI_DISPLAYS);
#endif
        KRN_DEBUG(1, "%i displays initialized", nr_displays);
```

Once all displays and focuses are created all maps can be initialized

```
        nr_focuses  = focus_init();

        kgi_init_maps(nr_displays, nr_focuses);
```

Finally intitialize all other KGI subsystems

```
        dev_console_init();
#ifdef CONFIG_KGI_DEV_GRAPHIC
        dev_graphic_init();
#endif
#ifdef CONFIG_KGI_DEV_EVENT
        dev_event_init();
#endif
        /* !!!  vcs_init();     */
}
```

1.4 Display Management

1.4.1 kgi_display_t structure

Each display present is represented by a *kgi_display_t*

```
typedef struct kgi_display_s kgi_display_t;

typedef void kgi_display_refcount_fn(kgi_display_t *);

typedef kgi_error_t kgi_display_check_mode_fn(kgi_display_t *dpy,
                kgi_timing_command_t cmd, kgi_image_mode_t *img, kgi_u_t images,
                void *dev_mode, const kgi_resource_t **r, kgi_u_t rsize);

typedef void kgi_display_set_mode_fn(kgi_display_t *dpy,
                kgi_image_mode_t *img, kgi_u_t images, void *dev_mode);

typedef kgi_error_t kgi_display_command_fn(kgi_display_t *dpy,
                kgi_u_t cmd, void *in, void **out, kgi_size_t *out_size);

struct kgi_display_s
{
        kgi_u_t         revision;       /* KGI/driver revision  */
        kgi_ascii_t     vendor[64];     /* manufacturer         */
        kgi_ascii_t     model[64];      /* model/trademark      */
        kgi_u32_t       flags;          /* special capabilities */
        kgi_u_t         mode_size;      /* private mode size    */


        kgi_mode_t      *mode;  /* currently set mode                   */

        kgi_u_t id;             /* display number, init to -1           */
        kgi_u_t graphic;        /* non-console devices attached         */
        struct kgi_display_s *prev; /* previous driver          */

        kgi_display_refcount_fn         *IncRefcount;
        kgi_display_refcount_fn         *DecRefcount;

        kgi_display_check_mode_fn       *CheckMode;
        kgi_display_set_mode_fn         *SetMode;
        kgi_display_set_mode_fn         *UnsetMode;
        kgi_display_command_fn          *Command;

        struct kgi_device_s     *focus;
};
```

revision

vendor, model are strings describing the display driver

flags

mode_size specifies the size of the display specific portion of the *kgi_mode_t* structure

mode is the currently set mode on the display. Mapping and unmapping a device will change the display's mode.

id is the id number of this display

graphic specifies whether the display is currently in graphic mode. Displays in graphics mode cannot be replaced.

prev points to the previous display (the one replaced by this display driver)

IncRefcount, DecRefcount are used to maintain reference count on displays (used when a display dirve is replaced by a different driver)

CheckMode is used to verify that a given mode is valid for this display. Also, it will fill out private section of the *kgi_mode_t* structure making the structure valid for setting mode on this display.

SetMode is used to actually set a mode (provided is has been checked by this display)

UnsetMode will unset te mode

Command is a method of sending generic commands to the display

1.4.2 kgi_register_display

The function *kgi_register_display* is used to replace a display with a new one. This function is mostly used by the KGIM subsystem to register a native driver loaded as a module.

```
/*      Register <dpy> under <id>. If <id> is negative, we search for the first
**      free one.
*/
kgi_error_t kgi_register_display(kgi_display_t *dpy, kgi_u_t id)
{
        kgi_display_t *prev;
        kgi_u_t i;

        KRN_DEBUG(2, "registering %s %s display with id %i",
                dpy->vendor, dpy->model, id);
```

Specifying negative id will autoassign the first available id

```
if (! KGI_VALID_DISPLAY_ID(id)) {

        for (id = 0; (id < KGI_MAX_NR_DISPLAYS) && kgidpy[id]; id++) {
        }
        if (KGI_MAX_NR_DISPLAYS <= id) {

                return -ENOMEM;
        }
        KRN_DEBUG(2, "auto-assigned new id %i", id);
}
```

A display with id has to exist, because KGI can only replace display. Also, the display must not be in graphic mode.

```
if (!KGI_VALID_DISPLAY_ID(id) || !dpy ||
        (kgidpy[id] && kgidpy[id]->graphic)) {

        KRN_DEBUG(0, "can't replace display %i", id);
        return -EINVAL;
}
```

Everything is fine so the display is initialized, and the pointer to the previous display is stored in the *prev* field. If anything goes wrong, or the display is unregistered, the previous display will be restored.

```
dpy->id = id;
dpy->graphic = 0;
dpy->prev = kgidpy[id];
kgidpy[dpy->id] = dpy;
```

If any of the devices registered on the display are console devices, but the display is unable to set the appropriate (text) mode, the new display cannot replace the old one.

```
if (kgi_must_do_console(dpy) && !kgi_can_do_console(dpy)) {

        KRN_DEBUG(1, "new display has to but can't do console");
        kgidpy[id] = dpy->prev;
        dpy->prev = NULL;
        dpy->id = -1;
        return -EINVAL;
}
```

Walk down the chain of replaced displays and increase their reference count

```
        prev = dpy->prev;
        while (prev) {

                (prev->IncRefcount)(prev);
                prev = prev->prev;
        }
```

Devices attached to the previous display most likely accessed some of the display resources. In order for them to use the new display's resources, they need to be reattached.

```
        for (i = 0; i < KGI_MAX_NR_DEVICES; i++) {

                if (display_map[i] == id) {

                        kgi_reattach_device(i);
                }
        }

        KRN_NOTICE("display %i: %s %s registered", dpy->id,
                dpy->vendor, dpy->model);
        return KGI_EOK;
}
```

1.4.3 kgi_unregister_display

Unregistering a display follows the same pattern as registering

```
void kgi_unregister_display(kgi_display_t *dpy)
{
        kgi_display_t *prev;
        kgi_u_t i;

        KRN_ASSERT(dpy);
        KRN_ASSERT(KGI_VALID_DISPLAY_ID(dpy->id));
        KRN_ASSERT(kgidpy[dpy->id] == dpy);
        KRN_ASSERT(dpy->graphic == 0);
```

The old display is reinstated

```
        kgidpy[dpy->id] = dpy->prev;
```

All devices are reattached in order for them to take into account the change of displays

```
        for (i = 0; i < KGI_MAX_NR_DEVICES; i++) {

                if (display_map[i] == dpy->id) {

                        kgi_reattach_device(i);
                }
        }
```

Finally walk down the list of replaced display and decrease their reference count.

```
        prev = kgidpy[dpy->id];
        while (prev) {

                (prev->DecRefcount)(prev);
                prev = prev->prev;
        }

        KRN_NOTICE("display %i: %s %s unregistered", dpy->id,
                dpy->vendor, dpy->model);
        dpy->id   = -1;
        dpy->prev = NULL;
}
```

1.4.4 kgi_must_do_console, kgi_can_do_console

Registration of a display required KGI to determine whether the display must be able to display a console. This is accomplished by scanning the list of all devices, isolating the ones registered to a given display. If any of these devices is a console, the new display must be able to display a console.

```
static inline kgi_u_t kgi_must_do_console(kgi_display_t *dpy)
{
        kgi_u_t i;

        for (i = 0; i < KGI_MAX_NR_DEVICES; i++) {

                if ((display_map[i] == dpy->id) && kgidevice[i] &&
                        (kgidevice[i]->flags & KGI_DF_CONSOLE)) {

                        return 1;
                }
        }
        return 0;
}
```

To see if a display can display a console, KGI attempts to set a **TEXT16** mode. If it succeeds the display is able to display a console.

```
static inline kgi_u_t kgi_can_do_console(kgi_display_t *dpy)
{
        kgi_mode_t mode;
        kgi_error_t err;

        memset(&mode, 0, sizeof(mode));
        mode.images = 1;
        mode.img[0].flags |= KGI_IF_TEXT16;

        err = kgidpy_check_mode(dpy, &mode, KGI_TC_PROPOSE);

        if (mode.dev_mode) {

                kfree(mode.dev_mode);
        }

        return (KGI_EOK == err) ? (mode.img[0].flags & KGI_IF_TEXT16) : 0;
}
```

1.5 Mode Setting

1.5.1 KGI Mode

In KGI a display mode is stored in the *kgi_mode_t* structure.

```
#define KGI_MODE_REVISION       0x00010000
typedef struct
{
        kgi_u_t                 revision;  /* data structure revision   */
        void                    *dev_mode; /* device dependent mode     */

        const kgi_resource_t    *resource[__KGI_MAX_NR_RESOURCES];

        kgi_u_t                 images;    /* number of images          */
        kgi_image_mode_t        img[1];    /* image(s)                  */

} kgi_mode_t;
```

revision is constant for backwards compatibility purposes

dev_mode is a pointer to display driver specific iformation for this mode. This data is allocated by KGI before a mode is checked with the display driver. The amount allocated depends on the *mode_size* in the *kgi_display_t* structure.

resources is an array of all resources available while the display is in this mode. This array is filled by the display driver during the checking of the mode

images specifies the number of images present in this mode

img specifies the mode for each image. Note that this field makes this structure's size flexible, the device needs to allocate enough memory for all image modes to fit

1.5.2 KGI Image and Image Mode

An image is an equivalent of an overlay plane. A mode can have any number of images, each with totally different characteristics specified by the *img* field.

The characteristics of each image are specified by the *kgi_image_mode_t* structure.

```
#define __KGI_MAX_NR_IMAGE_RESOURCES    16
typedef struct
{
        kgi_dot_port_mode_t *out;
        kgi_image_flags_t flags;

        kgi_ucoord_t    virt;
        kgi_ucoord_t    size;

        kgi_u8_t        frames;
        kgi_u8_t        tluts;
        kgi_u8_t        iluts;
        kgi_u8_t        aluts;
        kgi_attribute_mask_t    ilutm;
        kgi_attribute_mask_t    alutm;

        kgi_attribute_mask_t    fam, cam;
        kgi_u8_t                bpfa[__KGI_MAX_NR_ATTRIBUTES];
        kgi_u8_t                bpca[__KGI_MAX_NR_ATTRIBUTES];

        kgi_resource_t  *resource[__KGI_MAX_NR_IMAGE_RESOURCES];

} kgi_image_mode_t;
```

out specifies the dot port that this image will be displayed on (more on the concept of dot ports below)

flags specify various characteristics of the image

KGI_IF_ORIGIN origin can be changed

- **KGI_IF_VIRTUAL** virtual size can be changed
- **KGI_IF_VISIBLE** visible size can be changed
- **KGI_IF_TILE_X** can do virtual x tiling
- **KGI_IF_TILE_Y** can do virtual y tiling
- **KGI_IF_ILUT** can do index -¿ attribute mapping
- **KGI_IF_ALUT** can do attribute-¿attribute mapping
- **KGI_IF_TLUT** can do pixel texture (font) look-up
- **KGI_IF_STEREO** stereo image
- **KGI_IF_TEXT16** can do text16 output

virt specifies the virtual (total) size of the image

size specifies the visible size of the image

frames specifies the number of frames the image has (for example a double buffered image would have two frames)

tlut specifies the number of texture lookup tables (for font mapping)

ilut specifies the number of image lookup tables (for paletted modes)

alut specifies the number of attribute lookup tables (FIXME: what are these for?)

ilutm is a collection of attributes or'ed together specifying attributes that can be specified in the image lookup table

- **KGI_A_COLOR1** intensity of color channel 1
- **KGI_A_COLOR2** intensity of color channel 2
- **KGI_A_COLOR3** intensity of color channel 3
- **KGI_A_COLOR_INDEX** color index value
- **KGI_A_ALPHA** alpha value
- **KGI_A_PRIVATE** hardware or driver private
- **KGI_A_APPLICATION** store what you want here
- **KGI_A_STENCIL** stencil buffer
- **KGI_A_Z** z-value
- **KGI_A_FOREGROUND_INDEX** foreground color index
- **KGI_A_TEXTURE_INDEX** texture index
- **KGI_A_BLINK** blink bit/frequency

alutm provides similar mask for the attribute lookup table

fam is a list of attributes that are stored for each frame

cam is a list of attributes that are common to all frames for this image

bpfa is an array of sizes (in bits) for each of the attribute specified in the *fam* mask

bpca in an array of sizes (in bits) for each of the attribute specified in the *cam* mask

resource is a list of resources for this image. Image resources include image, attribute and texture lookup tables as well as text rendering interface

1.5.3 Dot Streams

Images are so called *dot stream sources*. They contain data which gets passed on to *dot stream sinks* such as monitors. The data can be converted along the way by a *dot stream converter* such as a DAC. Currently dot stream converters aren't used anywhere.

A dot stream sink is represented by an instance of *kgi_dot_port_mode_t* structure. Normally, there will be one such dot sink representing the monitor. This dot sink will be plugged into the *out* field of the *kgi_image_mode_t* structure. This establishes the connection between the monitor and the source of data displayed on the monitor.

```
typedef struct
{
        kgi_dot_port_flags_t    flags;  /* flags                */
        kgi_ucoord_t            dots;   /* image size in dots   */
        kgi_u_t                 dclk;   /* (max) dot clock      */
        kgi_ratio_t             lclk;   /* load clock per dclk  */
        kgi_ratio_t             rclk;   /* ref clock per dclk   */
        kgi_attribute_mask_t    dam;
        const kgi_u8_t          *bpda;

} kgi_dot_port_mode_t;
```

flags collection of attributes specifying dot port's mode of operation

- **KGI_DPF_CS_LIN_RGB** the dot port outputs data in linear RGB format
- **KGI_DPF_CS_LIN_BGR** the dot port outputs data in linear BGR format
- **KGI_DPF_CS_LIN_YUV** the dot port outputs data in linear YUV format
- **KGI_DPF_CH_ORIGIN** = 0x00000100, /* image orgin */
- **KGI_DPF_CH_SIZE** = 0x00000200, /* image size */

KGI_DPF_CH_ILUT the dot port is capable of changing the index – attribute mapping after the mode is set. This indicates the presence of an index lookup table. A paletted mode will have this.

KGI_DPF_CH_ALUT the dot port is capable of changing the mappings of attributes to different attributes. This indicates the presence of an attribute lookup table. Used mostly in text mode.

KGI_DPF_CH_TLUT the dot port is capable of changing the mapping between index values and the texture (text) appearing on the screen. This indicates the presence of a texture lookup table. Present in text modes.

KGI_DPF_TP_LRTB_I0 left to right, top to bottom, non-interlaced

KGI_DPF_TP_LRTB_I1 left to right, top to bottom, interlaced

KGI_DPF_TP_2XCLOCK load twice per cycle

dots indicates the size of the dot port. Note that this value is in dots, which are the actual pixels on the monitor, so for example for a text mode, this value would be 720x400 instead of 80x25.

dclk indicates the dot clock of the dot port. This is the speed at which the dot port outputs dots.

lclk is the ratio between dot clock and load clock. Load clock indicates how fast the dot port reads data from the associated image. For example, if the card has a 64-bit wide bus between the ramdac and the framebuffer and an 8-bit mode is set, *lclk* would be 1/8 since the dot port needs to do one load per 8 dots.

rclk FIXME

dam specifies the attributes that are present in the data form the associated image.

bpda specifies the sizes of each of the above attributes.

Dot stream converters can be used to connect different dot ports together. This is currently unused.

```
/*      A dot stream converter (DSC) reads data from it's input dot port(s),
**      performs a conversion on it and outputs the result on the output port.
*/
typedef struct
{
        kgi_dot_port_mode_t     *out;
        kgi_u_t                 inputs;
        kgi_dot_port_mode_t     in[1];

} kgi_dsc_mode_t;
```

1.5.4 Mode Checking

Before a mode can be set, it must first be checked by the display. This is handled by the *kgidpy_check_mode* function

```
/*      Check if a mode is valid. This returns a valid <mode> and KGI_EOK if
**      the mode can be done. NOTE: <cmd> must be either KGI_TC_PROPOSE or
**      KGI_TC_CHECK. <dpy> and <mode> must be valid.
*/
inline kgi_error_t kgidpy_check_mode(kgi_display_t *dpy, kgi_mode_t *m,
        kgi_timing_command_t cmd)
{
        kgi_error_t err;
        kgi_u_t i;

        KRN_ASSERT(dpy);
        KRN_ASSERT(m);
```

Mode checking is a collaborative procedure. A display can be composed of many subsystem and they all have to agree on the mode in general, and the video timings in particular. Mode checking is done in several passes, each pass having a different meaning depending on the *timing command*. There are four timing commands:

KGI_TC_PROPOSE the mode structure does not contain valid timings, the display (or its subsystems) are expected to produce the best suitable timings depending on the rest of mode parameters

KGI_TC_LOWER the mode structure contains mode timings, but the display (or its subsystems) are free to lower them if they are unable to handle the current ones

KGI_TC_RAISE the mode structure contains mode timings, but the display (or its subsystems) are free to raise them if they need to

KGI_TC_CHECK the mode structure contains mode timings that should not be changed. The display (and its subsystems) should merely verify that it can indeed handle the timings and prepare the display specific information for the mode to be set. This is always the last command in the timing negotiation sequence.

KGI initiates only the first pass of mode checking. It is expected that the display driver will do several passes in between its subsystems. Since this function initiates only the first pass, only **KGI_TC_PROPOSE** and **KGI_TC_CHECK** timing commands make sense.

```
        KRN_ASSERT(cmd == KGI_TC_PROPOSE || cmd == KGI_TC_CHECK);
```

If necessary, allocated the display specific mode structure pointed at by the *dev_mode* field in the *kgi_mode_t* structure.

```
	if (!m->dev_mode && dpy->mode_size) {

		m->dev_mode = kmalloc(dpy->mode_size, GFP_KERNEL);
		if (! m->dev_mode) {

			return -ENOMEM;
		}
	}
```

Call the display *CheckMode* function to do the actual mode checking. If the mode checks ok, the function will also fill in an array with all the resources available in this mode

```
	err = (dpy->CheckMode)(dpy, cmd,
		m->img, m->images, m->dev_mode,
		m->resource, __KGI_MAX_NR_RESOURCES);
```

Accelerator resources require special handling, the *idle* wait queue needs to be initialized

```
	for (i = 0; i < __KGI_MAX_NR_RESOURCES; i++) {

		if (m->resource[i] &&
			(m->resource[i]->type == KGI_RT_ACCELERATOR)) {

			kgi_accel_t *accel = (kgi_accel_t *) m->resource[i];
			wait_queue_head_t *idle;
			if (NULL == (idle = kmalloc(sizeof(*idle),
				GFP_KERNEL))) {

				KRN_DEBUG(1, "out of memory!");
				m->resource[i] = NULL;
			}
			accel->idle = idle;
			init_waitqueue_head(accel->idle);
		}
	}
```

If a problem occured, clean up and exit with an error code.

```
	if (err) {

		kfree(m->dev_mode);
```

```
                m->dev_mode = NULL;
                return err;
        }
        return KGI_EOK;
}
```

1.5.5 Mode Releasing

The *kgidpy_release_mode* function deallocates all data associated with a display mode

```
static inline void kgidpy_release_mode(kgi_display_t *dpy, kgi_mode_t *m)
{
        kgi_u_t i;
```

The accelerator's wait queue needs to be deallocated

```
        for (i = 0; i < __KGI_MAX_NR_RESOURCES; i++) {

                if (m->resource[i] &&
                        (m->resource[i]->type == KGI_RT_ACCELERATOR)) {

                        kgi_accel_t *accel = (kgi_accel_t *) m->resource[i];
                        kfree(accel->idle);
                        accel->idle = NULL;
                }
        }
```

As well as display's private mode structure

```
        if (m->dev_mode) {

                kfree(m->dev_mode);
                m->dev_mode = NULL;
        }
}
```

1.5.6 Mode Setting and Unsetting

When a device is mapped, the (previously checked) mode is actually set. This is handled by the *kgidpy_set_mode* function. All this function does is call the display's *SetMode* function.

```
static inline void kgidpy_set_mode(kgi_display_t *dpy, kgi_mode_t *m)
{
```

```
        KRN_ASSERT(dpy);
        KRN_ASSERT(m);
        KRN_ASSERT(dpy->mode_size ? m->dev_mode != NULL : 1);

        (dpy->SetMode)(dpy, m->img, m->images, m->dev_mode);
}
```

Upon unmapping of a device the mode needs to be unsed, which is handled by the *kgidpy_unset_mode* function. The actual action is done by the display's *UnsetMode* function.

```
static inline void kgidpy_unset_mode(kgi_display_t *dpy, kgi_mode_t *m)
{
        KRN_ASSERT(dpy);
        KRN_ASSERT(m);
        KRN_ASSERT(dpy->mode_size ? m->dev_mode != NULL : 1);

        if (dpy->UnsetMode) {

                (dpy->UnsetMode)(dpy, m->img, m->images, m->dev_mode);
        }
}
```

1.6 Device Management

1.6.1 kgi_device_t structure

Device is a particular view of a display. For example all vt's on the same head would be devices of the one display. Each display can have more than one device registered, but exactly one device is mapped to a display at a time. Each device maintains its own display mode that gets set every time the device is mapped to a display.

```
typedef enum
{
        KGI_DF_FOCUSED                  = 0x00000001,
        KGI_DF_CONSOLE                  = 0x00000002

} kgi_device_flags_t;

typedef void kgi_device_map_device_fn(struct kgi_device_s *);
typedef kgi_s_t kgi_device_unmap_device_fn(struct kgi_device_s *);
typedef void kgi_device_handle_event_fn(struct kgi_device_s *, kgi_event_t *);

typedef struct kgi_device_s
```

```
{
        kgi_u_t                 id;
        kgi_u_t                 dpy_id;
        kgi_device_flags_t      flags;

        kgi_device_map_device_fn        *MapDevice;
        kgi_device_unmap_device_fn      *UnmapDevice;
        kgi_device_handle_event_fn      *HandleEvent;

        kgi_mode_t      *mode;

        kgi_private_t   priv;

} kgi_device_t;
```

id is the device id

dpy_id is the id of the display this device is registered on

flags **KGI_DF_FOCUSED** means the device is currently mapped to its display

KGI_DF_CONSOLE means the device is a console device

MapDevice, UnmapDevice are callback that are called by KGI every time the device is mapped to or unmapped from its display

HandleEvent

mode points to the mode structure that will be set on the display every time the device is mapped to its display

priv can be used by the creator of the device to store its private data (the */dev/graphic* mapper uses it to store its data)

1.6.2 Device Registration

The function *kgi_register_device* will register the device *dev* as the device number *index*

```
kgi_error_t kgi_register_device(kgi_device_t *dev, kgi_u_t index)
{
        kgi_s_t err;
        kgi_u_t focus, console;
        kgi_u8_t *map;

        KRN_ASSERT(dev);
        if (! (dev && KGI_VALID_CONSOLE_ID(index))) {
```

```
                KRN_DEBUG(1, "invalid arguments %p, %i", dev, index);
                return -EINVAL;
        }
```

The actual number of the device depends on whether it is a console or not. The lower half of device ids is used for consoled, the upper half is used for graphic devices.

```
        dev->id = (dev->flags & KGI_DF_CONSOLE)
                ? index : index + KGI_MAX_NR_CONSOLES;
```

Pick the appropriate map depending on whether this device is console or graphic device.

```
        KRN_ASSERT(sizeof(console_map) == sizeof(graphic_map));
        map = (dev->flags & KGI_DF_CONSOLE) ? console_map[0] : graphic_map[0];
```

Now loop through the map looking for the spot where this device belongs

```
        index = 0;
        while ((index < sizeof(console_map)) && (map[index] != dev->id)) {

                index++;
        }
```

FIXME: THIS CAN'T BE RIGHT, (it should be 16)

```
        focus   = index / KGI_MAX_NR_CONSOLES;
        console = index % KGI_MAX_NR_CONSOLES;
```

Verify that this is a valid device id (the above loop found an appropriate spot in the map)

```
        if (! (KGI_VALID_FOCUS_ID(focus) && KGI_VALID_CONSOLE_ID(console) &&
                KGI_VALID_DEVICE_ID(map[index]) && (map[index] == dev->id))) {

                KRN_DEBUG(1, "no %s device allowed (device-id %i)",
                        (dev->flags & KGI_DF_CONSOLE) ? "console" : "graphic",
                        dev->id);
                dev->id = KGI_INVALID_DEVICE;
                return -ENODEV;
        }
```

It is an error to register two devices with the same id

```
if (kgidevice[dev->id]) {

        KRN_DEBUG(1, "device %i (%s %i-%i) is busy", dev->id,
                (dev->flags & KGI_DF_CONSOLE) ? "console" : "graphic",
                focus, console);
        dev->id = KGI_INVALID_DEVICE;
        return -EBUSY;
}
```

Initialize the device's display id using the global *display_map*

```
dev->dpy_id = display_map[dev->id];
```

Now make sure that there is a display for the device to register to

```
if (! (KGI_VALID_DISPLAY_ID(dev->dpy_id) && kgidpy[dev->dpy_id])) {

        KRN_DEBUG(1, "no display to attach device (dpy %i, dev %i)",
                dev->dpy_id, dev->id);
        dev->dpy_id = KGI_INVALID_DISPLAY;
        dev->id  = KGI_INVALID_DEVICE;
        return -ENODEV;
}
```

Upon registration the device is expected to have an appropriate mode set up in the *mode* field of the *kgi_device_t* structure. Here the mode is checked and if the display driver okay's it, the device is ready to be mapped (the mode is ready to be set)

```
if ((err = kgidpy_check_mode(kgidpy[dev->dpy_id], dev->mode,
        KGI_TC_PROPOSE))) {

        KRN_DEBUG(1, "initial mode check with dpy %i failed (err = %i)",
                dev->dpy_id, err);
        dev->dpy_id = KGI_INVALID_DISPLAY;
        dev->id = KGI_INVALID_DEVICE;
        return -EINVAL;
}
```

Graphic devices cause the display to be marked as being in graphic mode, and its reference count increased. Displays in graphic mode cannot be replaced nor unregistered

```
if (! (dev->flags & KGI_DF_CONSOLE)) {

        kgi_display_t *dpy = kgidpy[dev->dpy_id];
```

```
		while (dpy) {

			dpy->graphic++;
			(dpy->IncRefcount)(dpy);
			dpy = dpy->prev;
		}
	}
```

Finally insert the pointer to the device to the map mapping device ids to pointers to *kgi_device_t* structures

```
	kgidevice[dev->id] = dev;
	return KGI_EOK;
}
```

1.6.3 Device Unregistration

Unregistration of a device is done through the *kgi_unregister_device* function. This function assumes that the device being unregistered is not currently mapped (focused)

```
void kgi_unregister_device(kgi_device_t *dev)
{
	KRN_ASSERT(dev);
	KRN_ASSERT(KGI_VALID_DEVICE_ID(dev->id));
	KRN_ASSERT(dev == kgidevice[dev->id]);
	KRN_ASSERT(! (dev->flags & KGI_DF_FOCUSED));
```

Unregistration of graphic devices will restore the state of the display making it possible to replace or unregister the display

```
	if (! (dev->flags & KGI_DF_CONSOLE)) {

		kgi_display_t *dpy = kgidpy[dev->dpy_id];

		while (dpy) {

			(dpy->DecRefcount)(dpy);
			dpy->graphic--;
			dpy = dpy->prev;
		}
	}
```

Clear the global map

```
        kgidevice[dev->id] = NULL;
```

Finally the display mode is released and the device and display ids for this device are invalidated.

```
        kgidpy_release_mode(kgidpy[dev->dpy_id], dev->mode);

        dev->dpy_id = KGI_INVALID_DISPLAY;
        dev->id = KGI_INVALID_DEVICE;
}
```

1.6.4 Device Mapping

A device can be mapped to the display it is registered on using the *kgi_map_device* function.

```
void kgi_map_device(kgi_u_t dev_id)
{
        kgi_device_t *dev;
        kgi_display_t *dpy;

        if (! (KGI_VALID_DEVICE_ID(dev_id) && (dev = kgidevice[dev_id]) &&
                KGI_VALID_DISPLAY_ID(display_map[dev_id]) &&
                (dpy = kgidpy[display_map[dev_id]]))) {

                KRN_DEBUG(3, "no target or display for device %i, no map done",
                        dev_id);
                return;
        }
        KRN_ASSERT(dpy->focus == NULL);

        KRN_DEBUG(2, "mapping device %i on display %i", dev->id, dpy->id);
```

After some sanity checks, the device is mapped. The *focus* field of the display's *kgi_display_t* structure is made to point to the mapped device, and device's flags are updated indicating that the device is currently mapped

```
        dpy->focus = dev;
        dev->flags |= KGI_DF_FOCUSED;
```

The mode for this device is set on the display

```
        kgidpy_set_mode(dpy, dev->mode);
```

The creator of the device is informed through a callback that the device has been mapped to its display

```
	if (dev->MapDevice) {

		(dev->MapDevice)(dev);
	}
}
```

1.6.5 Device Unmapping

Unmapping of a device is handled through the *kgi_unmap_device* function.

```
kgi_error_t kgi_unmap_device(kgi_u_t dev_id)
{
	kgi_device_t *dev;
	kgi_display_t *dpy;

	KRN_ASSERT(KGI_VALID_DEVICE_ID(dev_id));
	KRN_ASSERT(kgidevice[dev_id]);
	KRN_ASSERT(KGI_VALID_DISPLAY_ID(display_map[dev_id]));

	if (! (KGI_VALID_DEVICE_ID(dev_id) && kgidevice[dev_id] &&
		KGI_VALID_DISPLAY_ID(display_map[dev_id]) &&
		(dpy = kgidpy[display_map[dev_id]]))) {

		KRN_DEBUG(1, "no target or display, no unmap done");
		return -EINVAL;
	}

	if (! (dev = dpy->focus)) {

		return KGI_EOK;
	}

	KRN_DEBUG(2, "unmapping device %i from display %i", dev->id, dpy->id);
```

After some checking to make sure that there is indeed a device to unmap, the device's creator is informed that the device is about to be unmapped. If the callback returns an error, the unmapping is aborted.

```
	if (dev->UnmapDevice) {

		kgi_error_t err;
		if ((err = dev->UnmapDevice(dev))) {

			return err;
		}
	}
```

Unset the display mode for this device

```
        kgidpy_unset_mode(dpy, dev->mode);
```

Update the display's structure noting that there's now no currently mapped device, and update the device structure flags clearing the focused flag.

```
        dpy->focus = NULL;
        dev->flags &= ~KGI_DF_FOCUSED;

        return KGI_EOK;
}
```

1.7 Resources

For each mode that the display can set, it exports a certain number of resources. Some resources are common to the display as a whole (such as the framebuffer or the accelerator), these are listed in the *resource* field of the *kgi_mode_t* structure. Other resources are specific to each image and are collected in the *resource* field of the appropriate *kgi_image_mode_t* structure.

Each resources contains the **__KGI_RESOURCE** macro as its first field effectively deriving from the *kgi_resource_t* base structure.

```
#define __KGI_RESOURCE  \
        void            *meta;          /* meta language object         */\
        void            *meta_io;       /* meta language I/O context    */\
        kgi_resource_type_t     type;   /* type ID                      */\
        kgi_protection_flags_t  prot;   /* protection info              */\
        const kgi_ascii_t       *name;  /* name/identifier              */

typedef struct
{
        __KGI_RESOURCE

} kgi_resource_t;
```

The following is a brief description of all available resources.

1.7.1 MMIO

MMIO is used to export a region of display's memory. In most cases this would be a linear aperture mapped somewhere in the system's memory but the resource is flexible enough to allow access paged memory apertures.

```
typedef struct kgi_mmio_region_s kgi_mmio_region_t;

typedef void kgi_mmio_set_offset_fn(kgi_mmio_region_t *r, kgi_size_t offset);

struct kgi_mmio_region_s
{
        __KGI_RESOURCE

        kgi_u_t access;                 /* access widths allowed        */
        kgi_u_t align;                  /* alignment requirements       */

        kgi_size_t      size;           /* total size of the region     */
        kgi_aperture_t  win;            /* window aperture              */

        kgi_size_t      offset;         /* window device-local address  */
        kgi_mmio_set_offset_fn  (*SetOffset);
};
```

access

align

size specifies the total size of the memory mapped region

win specifies the window that must be used to access the MMIO. This is useful for the cases when only a part of the total region can be accessed at a time (bank switching)

offset FIXME

SetOffset is used for specifying the the part of the total region that is accessible through the *win* aperture

1.7.2 Accelerator

Accelerator resource provides a means of executing a set of display card specific commands

```
typedef void kgi_accel_init_fn(struct kgi_accel_s *a, void *ctx);
typedef void kgi_accel_done_fn(struct kgi_accel_s *a, void *ctx);
typedef void kgi_accel_exec_fn(struct kgi_accel_s *a,
                kgi_accel_buffer_t *b);
typedef void kgi_accel_wait_fn(struct kgi_accel_s *a);

typedef enum
{
```

```
} kgi_accel_flags_t;

typedef struct kgi_accel_s
{
	__KGI_RESOURCE

	kgi_accel_flags_t	flags;	/* accelerator flags		*/
	kgi_u_t		buffers;	/* recommended number buffers	*/
	kgi_u_t		buffer_size;	/* recommended buffer size	*/

	void		*context;	/* current context		*/
	kgi_u_t		context_size;	/* context buffer size		*/

	struct {

		kgi_accel_buffer_t *queue;	/* buffers to execute	*/
		void *context;			/* current context	*/

	} execution;			/* dynamic state		*/

	kgi_wait_queue_t	idle;	/* wakeup when becoming idle	*/

	kgi_accel_init_fn	*Init;
	kgi_accel_done_fn	*Done;
	kgi_accel_exec_fn	*Exec;

} kgi_accel_t;
```

flags specifies accelerator's features

- **KGI_AF_DMA_BUFFERS** the accelerator uses DMA to exec buffers

buffers the accelerator recommended number of buffers that should be used with this accelerator

buffer_size the recommended size of each buffer

context points to the current context. See context management later on.

context_size size of context structure for this accelerator.

execution is a collection of information pertaining to execution of buffers

- **queue** is a queue of accelerator buffers that are waiting to be executed. When an accelerator receives a buffer for execution while the engine is busy, it will queue the buffer here.
- **context** current context

idle is a wait queue on which processes can sleep if they want to be woken up when the accelerator engine goes idle

Init is called once every time this accelerator is mapped by a process

Done is called when the process unmaps the accelerator

Exec is used for executing individual buffers

Every time an accelerator is mapped the mapper is expected to allocate a context buffer of size *context_size*. Passing this buffer to the *Init* function will make it valid for this accelerator. This context is then passed along with buffers to be executed and the accelerator will ensure that the buffer will be executed with this buffer loaded.

Accelerator commands are passed to the accelerator inside a *kgi_accel_buffer_t* structure through the *Exec* function in the *kgi_accel_t* structure.

```
typedef struct kgi_accel_buffer_s kgi_accel_buffer_t;

struct kgi_accel_buffer_s
{
        kgi_accel_buffer_t *next;       /* next of same mapping         */
        kgi_aperture_t  aperture;       /* buffer aperture location     */
        void            *context;       /* mapping context              */
        kgi_u_t         priority;       /* execution priority           */
        kgi_wait_queue_t executed;      /* wakeup when buffer executed  */

        struct {

                kgi_accel_state_t       state;  /* current buffer state */
                kgi_accel_buffer_t      *next;  /* next in exec queue   */
                kgi_size_t              size;   /* bytes to execute     */

        } execution;
};
```

next points to the next buffer in the circular list of buffer for a given mapping. Every time the accelerator is mapped, the mapper will create a number of accelerator buffers and link them into a circular buffer using this field.

aperture location of the actual data to be executed

context is the context that needs to be loaded while this buffer is executed

priority FIXME

executed a queue a process can sleep on if it wants to be woken up when the buffer has been executed

execution is a collection of state information used during execution of the buffer

state specifies what is the buffer currently being used for

KGI_AS_IDLE has nothing to do

KGI_AS_FILL gets filled by application

KGI_AS_QUEUED is queued for execution

KGI_AS_EXEC being executed

KGI_AS_WAIT wait for execution to finish

next points to the next buffer in the queue of buffers waiting to be executed. The accelerator will use the field to queue up a buffer if it is executed while the engine is still busy.

size is the size in bytes that should actually be executed. This can be less than the total size of the buffer.

1.7.3 Text16

```
typedef struct kgi_text16_s kgi_text16_t;
struct kgi_text16_s
{
        __KGI_RESOURCE

        kgi_ucoord_t    size;   /* visible text cells   */
        kgi_ucoord_t    virt;   /* virtual text cells   */
        kgi_ucoord_t    cell;   /* dots per text cell   */
        kgi_ucoord_t    font;   /* dots per font cell   */

        void (*PutText16)(kgi_text16_t *text16, kgi_u_t offset,
                const kgi_u16_t *text, kgi_u_t count);
};
```

1.7.4 TLUT

```
typedef struct kgi_tlut_s kgi_tlut_t;
struct kgi_tlut_s
{
        __KGI_RESOURCE

        void (*Select)(kgi_tlut_t *tlut, kgi_u_t table);
        void (*Set)(kgi_tlut_t *tlut, kgi_u_t table, kgi_u_t index,
                kgi_u_t count, const void *data);
};
```

1.7.5 Marker

```
typedef enum
{
        KGI_MM_TEXT16   = 0x00000001,
        KGI_MM_WINDOWS  = 0x00000002,
        KGI_MM_X11      = 0x00000004,
        KGI_MM_3COLOR   = 0x00000008

} kgi_marker_mode_t;

typedef struct kgi_marker_s kgi_marker_t;
struct kgi_marker_s
{
        __KGI_RESOURCE

        kgi_marker_mode_t       modes;  /* possible operation modes     */
        kgi_u8_t                shapes; /* number of shapes             */
        kgi_u8_coord_t          size;   /* pattern size                 */

        void (*SetMode)(kgi_marker_t *marker, kgi_marker_mode_t mode);
        void (*Select)(kgi_marker_t *marker, kgi_u_t shape);
        void (*SetShape)(kgi_marker_t *marker, kgi_u_t shape,
                kgi_u_t hot_x, kgi_u_t hot_y, const void *data,
                const kgi_rgb_color_t *color);

        void (*Show)(kgi_marker_t *marker, kgi_u_t x, kgi_u_t y);
        void (*Hide)(kgi_marker_t *marker);
        void (*Undo)(kgi_marker_t *marker);
};
```

1.7.6 CLUT, ILUT, ALUT

```
typedef struct kgi_clut_s kgi_ilut_t;
typedef struct kgi_clut_s kgi_alut_t;

typedef struct kgi_clut_s kgi_clut_t;
struct kgi_clut_s
{
        __KGI_RESOURCE

        void (*Select)(kgi_clut_t *lut, kgi_u_t table);
        void (*Set)(kgi_clut_t *lut, kgi_u_t table, kgi_u_t index,
                kgi_u_t count, kgi_attribute_mask_t am, const kgi_u16_t *data);
};
```