Assistance with writing graphics card drivers for the BeOS

(version 1.2, English)

A thesis by:

Rudolf Cornelissen
student HIO- deeltijd (college education)
2018 dec 28 - 2019 April 25 (Dutch original at 2003, June 6)

Contact info: User rudolfc @ https://discuss.haiku-os.org/

Table of contents:

1. Preface

There are a lot of software developers around the world, but only a few of those really know how to write a (stable) hardware driver. Drivers however are indispensable for every operating system out there. This is a problem which should be solved or at least be minimized somehow.

This document offers practical support for this purpose. It is meant for people who are interested in general information about what drivers are made of, and also for people interested in writing a driver themselves. The principles of writing a graphicscard driver are explained and some guidelines are layed out which should be followed. The process is viewed from the BeOS perspective, and for a large part as well from the opensource successor Haiku.

Because most software developers have little knowledge about hardware, some parts of the hardware is explained. Theoretical information with accompanying examples are given. Some things are explained in depth, while others are only just touched. At the back of this document a glossary has been included. This list mostly explains hardware related items.

In this document we talk about the parts of which the driver is made of and how these parts are intertwined with the BeOS. We look at the API and what it looks like, along with the driver interface below that. In order to do this we wil talk about the 'hooks' known to the various driverparts. In the end we will get to the 'flag' level in the driver interface, because these flags have a lot of influence on how the API works out.

Because in this document we are looking from the driver writer's perspective, we look extensively at the driverhooks and a bit less often at the API. The API is largely explained in existing online BeOS documentation: "The BeBook. In lovely HTML" [1]. For Haiku online documentation also exists [2]. The API- and driver interface information we do talk about in this document form an important extension to the BeBook since it explains things not found in this documentation. Some parts though are covered in the BeBook as well: These are mentioned here because these are specific to graphics drivers, or sometimes items need further explanation.

Graphics cards nowadays still more or less build upon the old VGA standard once defined. Also these cards still contain 'standard VGA' graphics modes. Because the original VGA standard still comes (partly) into play, documentation about the old standard VGA cards is still valuable while writing graphicscard drivers.
An excellent example of a book that contains valuable information is the "Programmer's Guide to the EGA and VGA Cards" [3]. Technical information about for example the PCI and AGP buses and an example of a 2D accelerator graphicscard can be found in "The indispensable PC hardware book" [4].

Please note that this document does not pretend to be fully complete or 100% correct. It's not doable to know all there is to know. Thankfully that's not needed anyhow to still be able to get nice results in the end.
Also please note that the successor Haiku might differ in behaviour when compared to t he BeOS. For instance currently no 2D acceleration is supported with Haiku.

Structure of this document:
In order to quickly show and clarify the structure of graphicscard drivers in the BeOS the top-down approach while explaining is used. First the more general stuff is explained after which we will dive deeper into matters. For instance first the API will be explained after which the more deeply rooted driver in the system will be described.

When writing a graphicscard driver for the BeOS, about 5% (or less) of the time will be spent to the kerneldriver and the rest of the time will be spent with the accelerant. Therefore most stuff to explain is about the accelerant which is why this takes up most of the space in this document.

Aside from the knowledge shared in this document, there's a plan of approach ('roadmap') attached at the end of the document which forms a decently workable method for writing and testing a graphicscard driver in the BeOS.

When someone is actually going to write a graphicscard driver, this roadmap will be the most important part of this document. Where needed this roadmap points to other parts in this document containing more background information about the step to take, or to related information about it.

1.1 Problem description

It is not easy to write graphicscard drivers. On top of that there's not enough specific info available for the BeOS to write a decent graphicsdriver without further ado.

Because I wanted to learn anyway I put a lot of time into researching all interfaces in the BeOS which come into play here. Also I looked extensively at the (graphics) hardware for it's influence on the way the software needs to control it. This process took some two years time in which 20 hours was spent on it per week. A big part of this time was used to research and test graphics hardware because manufacturers only provide very little useable information. Also tests were needed to gain extra information about the inner workings of the BeOS.

1.2 Graduation assignment

The knowledge I have gained is pretty specialized and is known with only very few other people (in the BeOS community). The assignment therefore is:
Share the gained knowledge about writing graphicscard drivers in the (BeOS) community and provide a step-by-step plan (roadmap) which can guide people while writing such a driver.

So: transfer of knowledge. This document will be published on the internet.

1.3 About the author

Ever since elementary school I am a real 'technician'. When I was about 10 years old I encountered an old transistor radio in the attic at my parents home. It was defective and I started experimenting with it. By chance I did something which brought the thing back to life: Since that moment I was 'hooked' on electronics.

After completing MTS/MBO (dutch technical highschool) and HTS/HBO (dutch Bachelor of Engineering, B Eng, specialized for electronics design) it turned out my interests started to shift from electronics to software engineering for hardware. That is actually very convenient, because as an electronics designer you have to work with microcontrollers and (industrial) PC's in designs a lot these days. In earlier times you had to work with 'rigid' (wired) logic where digital design was required, while later on all that stuff was replaced with microcontrollers (and / or with programmable logic array chips (i.e. FPGA's)).

All in all I wanted to know more about computers and technical computer science so I could work more well-founded in the area of interfacing between electronics and computers.

Currently I find studying the 'lowlevel side' of PC's very interesting. This is the part of the computer which connects or interfaces to (it's own inner) hardware. This means I like to work on controlling pheripheral devices inside and outside of the PC, like for example using the USB bus or using PCI-slot cards. I find it hard to get a decent grip on this kind of stuff however for two reasons:

- Manufacturers usually make it hard for you because they don't want to provide the needed lowlevel information that is needed for interfacing to their products via a driver you create yourself. Even more general 'readable' documentation for more common hardware is sometimes hard to find.
- Writing drivers means you'll operate in between software and hardware. Few people know how to work correctly in this field, since knowledge is needed of two seperate territories in order to succesfully create a stable driver.

Because I find it very intriguing to work in this specific field, I decided to do a extra study at the 'Hanzehogeschool Groningen' for computer science which is HIO/HBO (dutch Bachelor of Information and Communication Technology, B ICT, specialized for technical computer science).

So which parts of the HIO study are usefull in particular for the driver-writing field?

- The technical side. The curriculum from the third year of the program is very interesting: everything about JAVA. Not as a language, but as a means to explain various principles. Multithreading and the use of semaphores are spread across the system for instance: also inside drivers. Object oriented programming is interesting, because it's used for application software (on the BeOS), as well as in (some) drivers.
- And the lowlevel side. Knowledge about this is especially important for drivers. For a networkcard driver for instance the subject 'networks and communication' from the fourth year is indispensable. Knowledge about the Intel x86 architecture is very interesting as well: for instance handy when reverse engineering on PC's. You just might need to do that every now and then.

1.4 About the BeOS

The name 'BeOS' stands for 'The Be Operating System'. This commercial operating system was developed by Be, Inc. located in Menlo Park, California, USA. Be doesn't exist anymore because they didn't have the power to create enough marketshare in the OS market, (also) because of the powerfull position competitor Microsoft has in the world, and because of the way that company treats possible competitors.

The small part of the Be company that remained longer on the internet [5] was kept alive to wait for the results of the ongoing law suite at the time against Microsoft for the benefit of the shareholders.

According to many people the BeOS is a 'textbook example ' of an OS: straightforward by design. Still free of all 'legacy' problems other systems carry with them. With a well thought out API. The best of different other OS worlds combined, focused towards desktop and multimedia use. Optimal server functionality is not the main priority. The end user, who has to work with the PC as a tool, is. The BeOS 'just works'. Or not at all. Reproducably so. Simple.

Because of Be's unique approach, the (small) user community was very fond of this system. When Be decided to quit, the user community decided to rebuild this OS as an opensource system: The openBeOS project [6]. People did't have to start from scratch completely as parts of the OS like the Deskbar and Tracker were already opensourced earlier by Be themselves. Later the opensource project was renamed to Haiku [7]. This name was chosen because of the haiku style of network error messages that were generated by the BeOS's native web browser.

Aside from the openBeOS project, also a commercial attempt was made to continue the BeOS. This was done by the company named YellowTab [8]. They could work on this because they had some kind of contract with 'Be'. After a few releases they also had to stop with the OS development and the company ceased to exist.

The 'new BeOS' set out to be binary compatible with the 'old'version, so existing applications and drivers could still be used. As a start this of course has it's advantages. But new 'features' which will be created by (graphicscard) hardware developers need to be supported by the BeOS. New drivers and applications need to be developed. And while most developers know how to create applications, only few of them know how to create a driver.

These gaps need to be solved somehow. This is where documents like this one would be able to help out…

2018 update:
In the meantime Haiku R1 has reached beta-stage. Indeed it is binary compatible to old existing BeOS applications and also for (some) drivers if you install the gcc2 hybrid 32-bit release. The developers are looking and working beyond that now: there's a 64-bit release, work is being done for some ARM processor systems, and slowly the drivers are changing. Some nice multiplatform applications are running on Haiku like for instance LibreOffice and Lazarus software development IDE. Haiku supports much more memory, does USB very nicely, etc.

It will be very interesting to see what the future will bring for this operating system...

1.5 About graphiccard drivers

In general:
A driver is used to give applications running on a computer (including the OS) access to the hardware inside or attached to that computer. Drivers often provide complete functions which configure their hardware completely all at once. Because drivers are more or less part of the OS their reliability is of great importance for the stability of the system as a whole. For example drivers are needed for:
- networkcards;
- soundcards;
- modems;
- mouse and keyboard;
- graphicscards.

Because generating images requires huge amounts of data to be transported in very short time intervals (like for instance showing moving images in a movie), graphicscard-drivers are a special case among drivers.

The graphics hardware is a special case as well (in this case among the plug-in cards) because of the same reason. The PCI bus often is a bottleneck for graphicscards because of it's relatively low bandwidth: this is why the industry came up with the AGP bus. The single AGP slot in a system is kind of an extended PCI slot. AGP has a higher speed, it can deliver more energy to the card, sits closer to the system CPU, knows more operational modes and it has it's own dedicated controller. Multiple PCI slots usually share a single PCI controller, because of which the already rather low bandwidth also has te be shared between the mounted cards.

2018 update:
After AGP came PCIe (PCI-express), which is a much nicer high-bandwidth bus. This time there are upto 16 serial lanes per PCIe controller which can all be in parallel connected to a single (graphics) card, or be shared over multiple cards. The serial speed per lane is increasing per revision of the PCIe specification as well. Another interesting fact about PCIe is that the speed to- and from the card is the same (so symmetrical) which is not the case for the 'tweaked PCI' bus called AGP. The original PCI bus is symmetrical though.

Most graphicscards connect to all 16 lanes (or 8 if you share with a second graphicscard), but for instance a PCIe Matrox G550 connects to just a single serial lane. The reason here is that this is a relatively old, so slower type graphicscard which simply does not need more speed to do it's thing optimally. Sound- and network cards for instance also mostly need just a single PCIe lane.

BeOS graphicscard drivers:
While most BeOS drivers consist of just a single file: the kernel driver, a graphicscard driver consists of two parts: the kernel driver and the accelerant. Both parts consist of a single file.

The kernel driver is actually a 'standard' driver, while the accelerant is 'something new'. The accelerant runs in userspace executing the actual graphicsdriver commands, while the kernelspace driver sits only 'below it' to allow the accelerant access to the card hardware. At the startup of the graphicsdriver the kerneldriver 'passes' to the accelerant as many resources as it can so the accelerant can access these resources without having to 'go through' the kerneldriver, while the remaining resources can be accessed via the kerneldriver using relatively simple functions there. This way two important benefits are gained when compared to 'standard' kerneldrivers: speed and stability, both simply because the kerneldriver is barely needed.

Most drivers in the BeOS are still written in plain C (as opposed to using C++), probably because at this (driver) level most (hardware) things still operate in a sequential manner. Applications however are written in C++ so object oriented programming and multithreading kan be (simply) used.

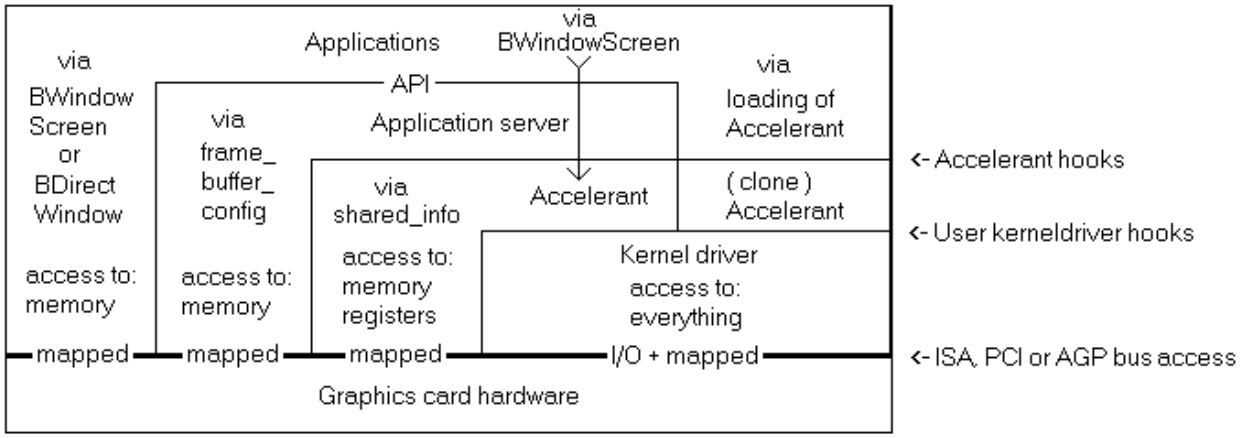
The BeOS (R5) supports the following settings in a graphics driver:
- set a display mode (i.e. resolution and refreshrate);
- hardware 'accelerated' panning and scrolling in virtual screens (by hardware pointer manipulation only);
- use of a hardware cursor (few colors);
- hardware accelerated 2D drawing (blitting, color-fills, inverts);
- hardware overlay ('offscreen' to 'onscreen' motion video, with colorspace conversion, scaling and filtering).

3D acceleration is (officially) not supported on BeOS R5, only software emulation exists for openGL. (There are two much used 3D drawing standards in the world: openGL is an open standard which is used on many 'platforms', while DirectX is a 'closed' standard belonging to Microsoft. DirectX is only used on Windows). Hardware accelerated openGL was planned for the next BeOS version, and only sits in 'dano', a once leaked version of the BeOS R5.1d0.

As a guide for testing and finding out why and how things work the way they do, the BeOS R4 Graphics Driver Kit [9] is used. This is the most recent known version of a description for graphicsdrivers from Be.

Access to BeOS graphicsdrivers

In the BeOS there are many ways to gain access to (parts of) the graphicscard. This has been done to be sure that there are options to gain maximum results in speed and simplicity for every situation that may arise. In the below picture the various ways to gain access to the hardware are shown.



Multiple ways to reach a single graphics card in BeOS R5

The most used path for applications or the app_server to gain access to the graphicscard is going through the primary loaded accelerant and the kerneldriver.
For special purposes other ways can be used:
- An application can gain direct access to the graphicscard's memory using the classes BWindowScreen or BDirectWindow;
- An application can gain direct access to the accelerant for 2D acceleration using BWindowScreen. The app_server will prevent it from using (some) other functions however;
- The app_server can gain direct access to the graphicscard's memory by using the struct frame_buffer_config. This struct can be fetched from the accelerant by using the hook GET_FRAME_BUFFER_CONFIG;
- The accelerant can directly access the graphicscard's memory and all memory-mapped registers. For the accelerant this direct route to the hardware is the path most used. The kerneldriver passes the required information for this via the struct shared_info. This struct is fetched by the accelerant during it's initializing phase using the kerneldriver user hook 'control';
- Aside from having (set-up) access to the graphicscard's memory and memory-mapped registers, the kerneldriver can also use I/O commands. With these the kerneldriver can access registers which cannot be mapped. The

possibility to map registers is determined by the card hardware. Older cards sometimes cannot map (all) registers, while newers cards (mostly) can.

A special way for an application to gain access to the graphicscard is to directly ‘privately’ load an accelerant. When for a specific card an accelerant is already loaded (by an application or the app_server), a ‘cloned’ accelerant will be automatically offered to the application which tries to load the accelerant. This clone has the same information as the original accelerant so they can use the same kerneldriver next to each other.

2. BeOS API classes for graphicscard drivers

In this chapter we will discuss the classes in the BeOS API which have influence on the operation of the graphicscard or vice versa. The information in this chapter is primarily meant for support while learning to understand how to write a graphicscard driver for the BeOS.

For normal use with the BeOS BScreen from the interface-kit is of great importance because this is the class that largely controls the driver.
For special purposes like games and multimedia two classes from the game-kit are important: BWindowScreen and BDirectWindow. With these classes direct and fast access to the card's framebuffer can be gained.
Last but not least you can use hardware overlay for video related applications. For this classes BView and Bbitmap are used (among others). These two classes will be described in more detail here in relation to the overlay functionality so it becomes clear how these things interact. Besides, the BeBook is incomplete and unclear about this subject.

2.1 BScreen (Interface-kit)

The most important class meant to control the graphicscard is BScreen. This class has the following functionality for applications:
- setting a graphics mode and supplying info about the currently active mode (screen resolution, colordepth, refreshrate, etc.),
- a mode proposition can be done which can be checked for validity (to be settable). If requested, it can be automatically adjusted to become valid if it's not fully OK. Note please that this function does not change the current active settings of the graphics card,
- in 8-bit depth mode you can set a colormap. This mode uses indexed colors which means that max. 256 seperate 24-bits depth colors will be selected and fetched from this map to be shown onscreen. So the 8-bit indexes are in the framebuffer each pointing at a location in that 8-bit indexed map for the 24-bit destination color to be shown for each specific pixel onscreen,
- vertical retrace synchronisation can be done which is used to prevent distortions onscreen if the content of the screen is changed (so to prevent tearing especially visible when a camera would pan left-right in for instance a movie scene),
- information about the graphicscard hardware can be obtained,
- mode timing limits can be obtained (so in effect the min. and max. settable refreshrates),
- the current DPMS state can be set and fetched (a monitor power-saving feature).

2.2 BWindowScreen (Game-kit)

BWindowScreen gives you total access to the entire screen content (so no visible windows at all remain). It bypasses the app_server completely for this. You have direct access to the driver: A display mode can be set from a predefined list, you can use the driver's acceleration and overlay funtions (if these exist in the current driver), and you can directly manipulate the framebuffer. Mouse and keyboard however remain under app_server control.
This class provides among others:
- Scrolling and panning in virtual screens (screenbuffer is larger than just the visual part) if supported by the driver;
- the accelerant hooks can be obtained and therefore they can be used directly (except for the cursor hooks: these are shielded here because the app_server remains controlling mouse and keyboard input);
- information about the currently active mode can be obtained.

2018 update:

Unfortunately on Haiku Scrolling and panning doesn't work as intended because of the way the native screen preferences app communicates with the drivers concerning dualhead support. The visible-window-in-the-total-screen pointer is being tweaked for this dualhead support tweak which should not be.

2.3 BDirectWindow(Game-kit)

Class BDirectWindow also offers applications direct access to the framebuffer on the graphics card. BdirectWindow however can be used in fullscreen mode and in windowed mode: Windowed mode cannot be done with BWindowScreen.
You can create a BDirectWindow which looks like a normal window, only now the application can draw in it via directly accessing the framebuffer. With normal windows (class BWindow) this is not possible. In this case the BeOS API is used to draw if needed.

BDirectWindow cannot do anything with the graphicscard, apart from offering direct access to the framebuffer to the user application. Things like setting a display mode are not possible. The driver however does have influence on the way BDirectWindow works. That's the reason this class is mentioned here.
At the description of function BDirectWindow.SupportsWindowMode() in the BeBook it says that the availability of windowed mode depends on the graphicscard hardware.
The BeBook says that useability of windowed mode depends on:
- hardware cursor support,
- DMA support and
- parallel access to the card hardware.

The mode.flags B_IO_FB_NA and/or B_PARALLEL_ACCESS probably have influence on how class BdirectWindow works in this respect.

2.4 Classes used for hardware overlay: BBitmap (Interface-kit)

A BBitmap is created to retain the data that is used for showing in a BView. In a BBitmap its memory can be directly accessed, so data can be drawn pixel by pixel directly into the bitmap: for showing moving video this is a fast method. When constructing a BBitmap the type of bitmap it will be is determined by the user program: an overlay- or a regular bitmap.
An overlay bitmap is created directly inside the graphicscard's memory, but outside of the part of this memory that is visible on the screen, while a regular bitmap is created inside the computer system's main memory. Overlay bitmaps are not created by the app_server itself, but via a function for that in the graphicscard driver.

The advantage of a bitmap residing inside the graphicscard memory is that this bitmap does not per se has to be copied to be shown onto the screen: changing the pointer to the start of the framebuffer in the VGA chip is enough to arrange for this in theory. A bitmap residing inside main memory however does need to get copied into the graphicscard memory first (using DMA, direct memory access, or another method) before it can be shown onto the screen.

The above information should make it clear to you that correctly choosing the kind of bitmap to use has great influence on the CPU load that is generated when showing a picture onscreen.

When constructing a bitmap the colorspace that will be used inside it needs to be selected. For plain (so non-overlay) bitmaps this will usually be the colorspace being currently active onscreen: so for example B_CMAP8, B_RGB16 of B_RGB32 (Intel architecture: little endian). Overlay bitmaps usually make use of B_YCbCr422 (registered by Nvidia as industry standard space YUY2[10]) because this colorspace is usually supported by graphicscards. Sometimes also B_YCbCr411 is used for example. The latter colorspace has a higher compressionrate (less bits/pixel on average) and therefore creates less CPU load on handling it.

The overlay engine inside the graphicscard can colorspace convert, scale and interpolate (filter) the bitmap (all upon request). The unit's output is always created in maximum colordepth (24bit or even more), so the user can fully enjoy the wealth of colors in a video while the (rest of the) desktop is shown in only 8bit colordepth for example. This type of configuration is possible because the overlay unit's output is not written back into the graphics memory, but instead is fed directly into the DACs (or digital datalink) towards the screen.

When a BBitmap is constructed to be an overlay bitmap, it's properties will be restricted by the graphicscard and it's driver.
These restrictions consists of minimal- and maximal scalingfactors useable when showing the bitmap using overlay (in a window) onscreen, minimal and maximal sizes of the bitmap itself, and a 'stepping size' (granularity) for the X-coordinate of the bitmap (the width of the bitmap must be divisible by the granularity).

When this last demand is not met when constructing the bitmap (the application normally does not know beforehand which restrictions might exist), then the driver automatically creates 'slopspace' inside the bitmap. Slopspace is extra, non-used space 'at the end of every line in the bitmap' which makes sure the resulting bitmap adheres to the restrictions that apply for this graphicscard.
Note please that this type of restrictions exist because hardware designers always strive for maximum performance. In order to process a bitmap with maximum speed it's memory needs to be read with specific minimal 'chunks' at once. Such a minimal chunk would usually correspond with the memory buswidth available to the GPU / graphicscard memory hardware interface. Also please note explicitly that the slopspace needed depends on memory buswidth, so not on a specific number of pixels. Depending on selected colorspace for the bitmap slopspace may therefore vary.

One final remark here: The slopspace thing you see being needed for hardware overlay (might) also exist for (setting) graphicsmodes in graphicscards for the same performance reasons. Nvidia cards for example are notorious for this, having relatively high granularity restrictions. The bytes per row item is therefore especially noteworthy also for graphics modes.

After the bitmap was constructed all this restriction information (so the overlay restrictions) can be fetched using a specific function of that bitmap. Also the number of bytes per row (so for the de X-coordinate) can be fetched. The number of returned bytes per row include the extra bytes needed as slopspace.

2.5 Classes used for hardware overlay: BView (Interface-kit)

BView is used to draw in a BWindow. Bview includes specific functionality for hardware overlay: SetViewOverlay() and ClearViewOverlay(). When an application is written for overlay use, it's handy to know that in the same way functions SetViewBitmap() and ClearViewBitmap() can be used if overlay is not possible on a graphicscard at all, or it's not possible in the currently active graphics mode, if you don't want to use it, or if the driver simply does not have the overlay functionality implemented.
Possible reasons for certain modes not being able to use overlay are for instance: lack of memory space or lack of 'spare' transferring (memory) bandwidth because we are currently in a too high resolution graphics mode (or too high a colorspace: note that twice the depth usually means twice the needed transferring bandwidth to fetch all the needed data from the graphics memory!).
Lack of memory can also be a reason why in certain high-res (or virtual scan/pan) modes an overlay application would use single buffered overlay as a fallback for doublebuffered overlay, still being much nicer (i.e. in CPU load or quality of picture on scaling) than even falling back to bitmap output mode..

SetviewOverlay is used to switch-on the overlay unit while presenting an overlay bitmap to use that was constructed earlier. This function kan also be used to switch between various overlay bitmaps (often three bitmaps are used for the so called 'double buffering' technique: Every movie frame the buffer to use is switched here, three (not two) buffers are used to bring the user the lowest CPU load possible). Note please that, while seldomly done, these bitmaps may have different sizes and colorspaces in theory.

With SetViewOverlay you can also specify if the complete input bitmap should be shown or that only a part of it should be shown in the output window. You can think of this as being a 'hardware zoom'facility.

SetViewOverlay after calling returns a so called ‘magic color’, which is used as a key (‘colorkey’) to determine if for each pixel onscreen (inside the overlay output area) the overlay unit’s output should be shown, or some other object being in front of (part of) the output window. The key is one specific color which is used by the graphicscard hardware to switch between both sources: Being overlayed video or desktop content (like a menu in front of a video or for instance another window being in front).
The returned colorkey should always be drawn by the application in the whole (complete) output window which will contain the overlayed video (if that is the application’s intention). This instructs the graphicscard hardware to show the output of the overlay unit instead of part of the framebuffer here. If a menu or so is shown on top of the output window by the app_server, it will not show this colorkey in that part of the output window and you will ‘magically’ see the menu or so just as you’d expect. The colorkey is determined by the app_server and it will pass this key to both the driver and the application.

Please note here that the app_server will (should) come up with a color otherwise not used by it (or by running applications), or at least be minimally used: so chances of unintended behaviour will be minimal. Also please note that if a user tries to make a screenshot of overlayed video, the magic color will be captured instead as a screenshot is taken from the framebuffer memory. If the user would then switch to another workspace, and look at the captured image there, video from the output window on the other workspace will be shown here in these parts of the shot that by chance overlap with the overlayed video window output position. This is not a fault, but a limitation of the overlay technique as you will understand. Another limitation is that only one video can be shown simultaneously per overlay unit (more than one unit can be built into a single graphicscard, though this is not done often).

Class BView’s function ClearViewOverlay at last will disable the current application’s use of overlay in the graphicscard hardware. It’s used when the application is done using overlay (for now).

2.6 Conclusion

Be apparantly strived to create an API for use of the graphicscard (driver) that is straightforward and easy to use. This also shows from the fact that only the more common built-in functionality of graphicscard hardware is supported.
After studying the accelerant hooks and hardware restrictions further down in this document we will find that some functionality is missing that would be needed for more complete (better) support of the functions that exist inside the API. The missing functionality can be partly explained by some accelerant functions still being experimental in the BeOS R5, like hardware overlay. BwindowScreen is not complete in the sense that there’s only a fixed list of graphics modes available while you should also be able to simply set a custom mode like you can do by using Bscreen’s SetMode function.

Even though the API is not fully finalized for existing functions yet it does get the job done already very nicely.

In the future support for things like TV video output, dualhead and DDC would be very welcome: common built-in functionality inside graphicscards is ever expanding with each new generation of cards that gets introduced.

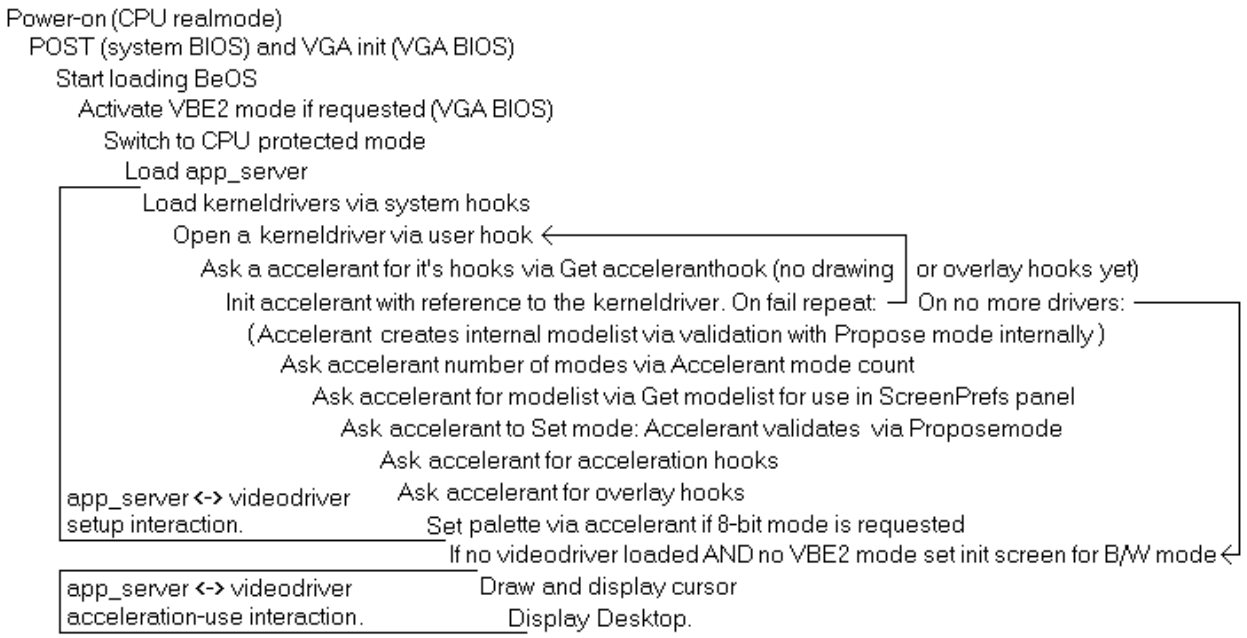
2018 update:
In the meantime TV video output is (more or less) nolonger needed since TVsets became much more like computer screens, DDC support (DDC is a serial communication channel between graphicscard and screen used to inform the computer about the screen’s specs mostly) has been built into Haiku and the graphicscard drivers, while dualhead support is still lacking in it’s app_server. All in all: Haiku is evolving, and hardware also keeps on evolving..

3. Kernel driver

The part of the graphicsdriver running in kernelspace is just there to give the accelerant running in userspace access to the graphicscard hardware. When use of the driver is started as many resources as is possible are 'passed through' to the accelerant so it can use the hardware without further needing the kerneldriver. Possible remaining resources that cannot be passed through to the accelerant are made accessible via simple functions inside the kerneldriver. This way two important advantages are gained in this 'split' driver setup when compared to normal single kerneldrivers: optimal speed and optimal stability, both because the kerneldriver is barely needed.
Optimal speed is gained because almost no costly context switches are needed between user- and kernelspace.
Stability is optimal because a crashing application in userspace won't take down the whole system, while this could very well be happening with a crash in kernelspace.

The kerneldriver contains two interfaces:
- an interface to the OS that takes care of initializing the driver and publishing it's name,
- an interface to users (like the accelerant) for using the driver for the hardware related functions.

Below picture indicating the starting sequence of the BeOS (in relation to graphicscard drivers) shows among other things how the kerneldriver is started. First all kerneldrivers will be initialized via de 'system hooks'. Those drivers that are succesfully initialized (these detected hardware they are supporting) can be used after that via the 'user hooks'.



Basic BeOS boot sequence in relation to videodrivers

After the kerneldrivers are initialized the first one that succeeded for graphicscards will be tried with all existing accelerants via its user interface. This process keeps repeating until an accelerant is found that reports via B_OK status it can work with the current kerneldriver. After this the rest of the system is loaded until we see our desktop onscreen.

In this chapter we will look at the kerneldriver's system- and userinterface. The next chapter covers the accelerant.

3.1 Interface to the OS

A kerneldriver will generally speaking be initialized by the OS during system start. After that it's available to the 'users'. For the initialisation phase each kerneldriver has a number of standarized routines:
- init_hardware;
- init_driver;
- uninit_driver;
- find_device;
- publish_devices.

Kerneldrivers must be at a specific place on disk. They belong in:
/boot/beos/system/add-ons/kernel/drivers/bin/ (for drivers delivered default with the system) or:
/boot/home/config/add-ons/kernel/drivers/bin/ (for (optional) drivers that are added later on).

In the first case the drivers have a symbolic link pointing to them in:
/boot/beos/system/add-ons/kernel/drivers/dev/, and in the latter case in:
/boot/home/config/add-ons/kernel/drivers/dev/.
In these paths the last part is /dev/ because 'dev' is the place directly under the root folder where in the end the name will be published which the driver exports. A later-on added kerneldriver for a graphicscard for example needs a link in: /boot/home/config/add-ons/kernel/drivers/dev/graphics/.

When problems arise with add-on drivers one can call an early configuration screen during boot (by pressing <space>) which belongs to the BeOS. When 'disable user-addons' is selected there the system won't load the later-on added drivers. When no driver for the current graphicscard remains and it's the only graphicscard in the system the BeOS will boot-up in black-and-white failsave graphics mode. Trouble can then be solved.

2018 update:
Haiku has more extensive options to select from during boot. Also Haiku has very nice VESA graphics mode support so even without a graphicsdriver the 'first' graphicscard in the system will always show the desktop in beautifull color.

The remainder of this chapter covers the kerneldriver hooks to the OS and to the user.

3.1.1. init_hardware

When a driver is installed correctly, the OS will init the driver during boot:
The first hook that gets called is init_hardware. This routine checks if the needed bus (like ISA and/or PCI) is available, and of course it checks if at least one supported device is available.

3.1.2 init_driver

When init_hardware was succesfull then init_driver is called. Here some needed variables will be initialized, like the name(s) of the device or devices found. Also the bus(ses) needed for access to the hardware will be opened for use here.

3.1.3 publish_devices

The system wil now ask for the name(s) of the supported hardware via hook publish_devices. The names will be published as symbolic links in the filesystem, below the /dev/ folder. Where exactly they are placed depends upon the place in the filesystem where the link to the kerneldriver resides, so that depends upon how the driver was installed (for more details see section 3.1 above). The driver is supposed to publish names for the hardware which are unique for each card in the system. Usually you will find the card's manufacturer ID and card ID (from PCI config space), along with a busnumber, slotnumber and possibly a function number. The bus- and slotnumber are depending

on in which slot inside the computer the card hardware was placed, so even two identical graphicscards will have different published devicenames in a system.
The kerneldriver is loaded just once. Such a driver supports a specific maximum number of cards being used simultaneously. The maximum number of cards it supports is determined in the driver's code.
A single unified driver (a single driver which supports multiple different graphicscard types) can for instance be opened three times. For example one time for an AGP slot card, and two times for two PCI cards. This will result in three different entries inside the /dev/ folder hierarchy. For a more detailed explanation please also see chapter 4.

So every 'filename' in the /dev/ hierarchy represents a supported existing device in or connected to this computer: a device which can actually be used on the specific system that has it's name published. When in the beginning of the process the driver's init_hardware hook wasn't succesfull, no names from this driver will be published in the /dev/ folder hierarchy.

Whenever a second accelerant (in case of a graphicscard driver) wants to control the same card hardware, the kerneldriver will already be opened and it will only track this using a 'times opened' counter: The driver and it's card hardware were already initialized before after all.

3.1.4 uninit_driver

Whenever a kerneldriver is nolonger used (with graphicscard drivers this will be when the last accelerant instance closes the driver), the OS will call uninit_driver. This routine should release all used (hardware) resources. Opened buses for instance need to be closed then.

3.1.5 find_device

Find_device at last returns the adresses (entry points) of all the routines (as a table) defined to be the standard interface for the driver-user. These driver_hooks are used later-on to access the driver to make use of it.

3.2 Interface to the user

Once the kerneldriver is initialized by the OS it can be accessed using the following hooks (among others):
- open_hook;
- close_hook;
- free_hook;
- control_hook;
- read_hook;
- write_hook.

With the above hooks the driver can be opened and closed, data can be read and written, and there's a control hook. The hooks are based on the file access system: A driver can for example be opened by opening the /dev/ name of the driver in question as if it were a file.

Application of a kerneldriver which is part of a graphicsdriver:
For a graphicsdriver only the open-, close- and control-hooks are of importance: the other hooks are not used. (A storage driver like a SCSI driver for example would use the read- and write-hooks to read and write data.)

For a graphicsdriver the accelerant is 'user'. For other drivers this can be for example some application directly. (With graphicsdrivers an application would use the accelerant when direct access to the graphicscard is needed).

The control hook is used to determine an accelerant which tries to use the kernel driver indeed has the rights to do so (they need to belong together). Furthermore I/O commands can be issued here for the ISA or PCI bus for example, and a private struct ('shared info') wil be passed to the accelerant so it has the graphicscard data it needs to control the target graphicscard (correctly).

The kerneldriver's function is recognizing supported graphicscards, 'map' them in the system memory, and make the ISA and 'PCI configuration space' I/O commands available if needed (these can only be executed in kernel space). When interrupts are needed they also run through the kerneldriver.

Resources in normal PCI I/O space can be mapped to system memory. All PCI(AGP) graphicscards support mapping the card's memory in the same way, and for most of them this applies also for (part of) the GPU registers. This concerns normal PCI registers (non-configspace) then. Mapping these resources causes that the accelerant, the OS and the applications can all access these resources without needing to go through the kerneldriver (speed gain).

3.2.1 open_hook

The open hook creates and initializes the 'shared info' struct which can be used by the kerneldriver itself, but also by all accelerants (both the original instance and all clones). This struct can hold things like the manufacturer- and card-ID's, the specification of the card hardware (i.e. max. DAC speed for every colordepth, memory size and speed, GPU core speed..) and for instance the current active display mode.
Furthermore the graphicscard is 'mapped' in memory as far as is possible and is needed. The framebuffer, PCI I/O registers, ROM and sometimes a pseudo-DMA area are candidates for this. Mapping the card has the advantage (as explained before) of later-on not needing to access the kerneldriver in order to access these items.

If possible also a semaphore is set-up which the accelerant can use later-on to synchronize update actions to the screen's vertical retrace (to prevent tearing).
Last but not least, an interrupt handler can be installed which for example handles stuff that needs to be done during vertical retrace, like unblocking the retrace semaphore. Ever since PR2 (preview 2) the BeOS automatically installs the handler in such a way that in case of a shared interrupt line (this happens rather often with PCI cards) all routines of all drivers are 'daisy-chained'. By use of a status these routines return to the BeOS, they can signal to it that an interrupt was handled and so this particular interrupt in this case was (indeed) meant for them.

3.2.2 close_hook

The close hook theoretically closes the kerneldriver. According to the BeOS documentation this hook does nothing with graphicsdrivers: it just doesn't return an error status here (return B_NO_ERROR).

3.2.3 free_hook

This hook actually closes down the driver. The number of times the driver was opened has been counted. If the free_hook is called for the last time, the driver is shutdown. This means all initialisation done at the open_hook will be undone now (in reverse order):
The card's interrupt sources will be disabled, pending interrupts are then removed. After this the interrupt routine is de-installed. Now the retrace semaphore is removed and the card's mapped areas will be removed. The shared info struct is destroyed.

3.2.4 control_hook

As mentioned earlier the control_hook is used to determine if an accelerant trying to use the kerneldriver has the authority to do so (they need to belong together). This is done via the kerneldriver's control_hook 'sub' function B_GET_ACCELERANT_SIGNATURE. This is the only 'public' function.

Furthermore I/O commands can be issued for ISA or PCI bus for example (like accessing PCI configuration space), and a private struct (with shared_info) can be obtained by the accelerant so it has all needed information about the graphicscard which it (and all the other accelerant copies, the 'clones') control.

Also for example interrupts can be enabled or disabled using the control_hook.

3.2.5 read_hook

The read_hook is unused. It only returns the B_NOT_ALLOWED status to indicate this.

3.2.6 write_hook

The write_hook is unused. It only returns the B_NOT_ALLOWED status to indicate this.

3.3 Conclusion

The kerneldriver interface is well thought out. This is not really surprising, since this interface already exists for a long time. Apart from that a larger part of this interface was borrowed from Linux, where this interface already existed for an even longer time.

The construction is not really exciting for graphicscard drivers, because the real functionality is built into the accelerant. Although care should be taken when writing a kerneldriver since a simple single fault can crash the whole operating system.

4. Accelerant

As opposed to the kerneldriver, the accelerant runs in user space. The accelerant provides functions that are needed to control a graphicscard. These functions are used by the app_server and/or applications directly.

There are a number of reasons for the graphicsdriver being divided into a kernel- and userspace part:
- speed: When controlling the graphicscard configuration (so programming the ‘registers’) is done using memory mapped I/O this can be done using pointers. This works much faster than doing a single I/O in kernelspace per register access. The context switch which has to be made then (twice per single I/O!) costs a lot of time. Especially when tens to hundreds of accesses need to be done this will slow things down considerably. (In theory the speed can be increased by sending lists of registers to the kerneldriver at once as this will minimize the number of context swiches, but this adds complexity considerably also.)
- security: A user program cannot make the system crash: a kernel space program however can. If the accelerant messes things up the app_server might go down. However access to the system remains functional using for instance a network connection to the system with telnet or ftp.

At least one instance of the accelerant is loaded for each published name from the kerneldriver. This way a single kerneldriver can support multiple cards it detected simultaneously. All loaded accelerants control their own graphicscard using communication with that single instance of the kerneldriver. The accelerant/driver ‘sets’ all work independently: variables inside the kerneldriver need to be seperated for each supported device (using arrays for example).

When needed, a second or later instance of the accelerant (called a 'clone') can be loaded for a graphicscard for which already a first ('original') accelerant was loaded. A clone has the same information (called ‘shared info’) as the original accelerant. 'Shared info' was created by the kerneldriver to be later on shared with all accelerants using this kerneldriver for the graphicscard this shared info belongs to. The original accelerant largely fills in the shared info it got from the kerneldriver. Via the kernel driver this information is also visible for all clones for the same graphicscard the original accelerant is controlling. Both original accelerant and clones create their 'own' cloned 'copy' of the shared info so they can all access it in their own adress space. When something is read from, or written to this cloned area actually the original shared info as created by the kerneldriver is accessed: which is the way all driver parts (kerneldriver, original- and cloned accelerants) can control and use the same shared information.

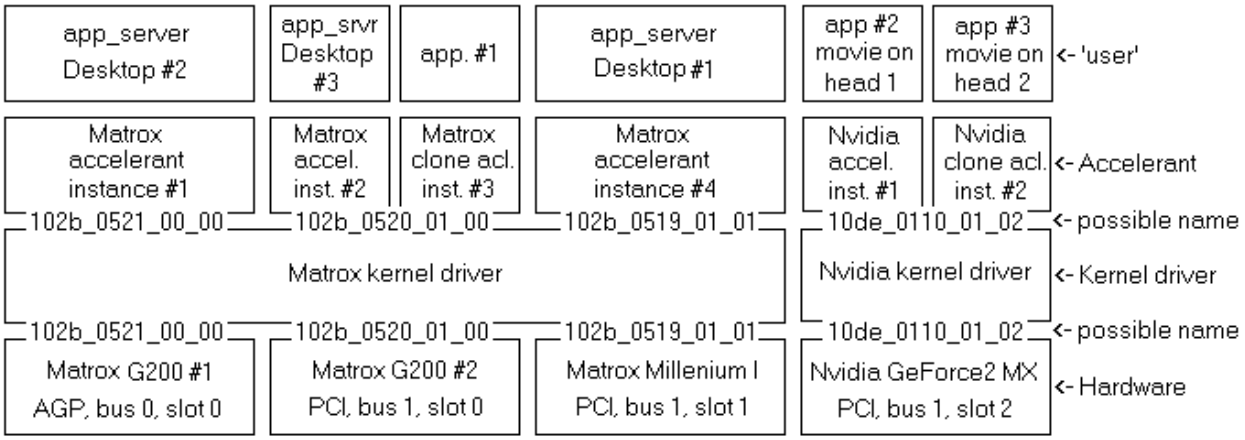
When a second kerneldriver would be loaded for another graphicscard: so that card is _not_ supported by the first kerneldriver, in theory a matching accelerant would be searched for by the system.

When the system is booted, a kerneldriver will be loaded for each and every graphicscard found in the system (if all cards have a BeOS driver that is). But because the BeOS currently only supports a single graphicscard with a single screen per system, only one 'original' accelerant will be loaded by the app_server for a 'randomly' chosen loaded graphics kerneldriver. Not entirely random though, it selects the first found kerneldriver via their more or less alphabetically ordered exported names.

An accerant clone is not much used in practice. A clone could for instance be used for controlling a second screen on the first single graphicscard in a system, for instance by a video consumer node: used for independent showing motion video on a reserved videoport on the graphicscard.

The application server ('app_server') is the BeOS part which among other things loads the accelerant and sets up the workspace. A second instance of the app_server in theory could load a clone for the first graphicscard, or a original accelerant for a second graphicscard, and show a workspace on a second screen.

The figure below shows another example of a theoretically possible configuration.



Theoretical possible BeOS setup with multiple graphics cards

In this picture you can see an example configuration with four graphicscards is a system: One AGP card and three PCI cards. The three Matrox cards are all supported (and so: controlled) by the same instance of one single kerneldriver. Difference is made via the exported names setup for the cards which all contain the manufacturer ID, card ID, system's busnumber, and slotnumber in the bus.

For the Matrox cards four instances of the exact same accelerant are loaded, while for the Nvidia card two instances of the belonging accelerant are loaded. (Theoretically even different accelerants could be used for different cards supported by a single kerneldriver.)

In this case three different desktops are shown on the three Matrox cards, while the Nvidia card is only used for showing two different fullscreen movies simultaneously. One on each screen of this dualhead kaart (Nvidia calls this 'twinview' on the card used here). The applications showing these movies could be video consumer nodes here for example. Furthermode one application has loaded an accelerant clone for the PCI G200, probably for tweaking or showing additional info to the user about the state the card is in. A very usefull example could be reading the fan speeds, temperatures and voltages on the card, though there is no accelerant hook defined (yet) which supports collecting this type of info from the card.

Note to this chapter:
Extra notes have been attached to the descriptions of various accelerant functions further down this chapter. These notes concern experiences gained by the author when working with these functions. Sometimes incompleteness or defects of the API are mentioned here. Also errors or incompleteness in the BeOS R4 Graphics Driver Kit are discussed here.

4.1 The accelerant hooks

Hooks that are not implemented in the accelerant are not exported by the accelerant. Instead it returns a null pointer for every unsupported hook.
Beware: Except where noted, hooks will not be requested again after a modeswitch!

One of the two hooks B_INIT_ACCELERANT or B_CLONE_ACCELERANT is requested and executed by the app_server (or application) before any other hook will be requested. All other ‘feature’ hooks can be existing or non-existing (so: null) depending on variable values during the accelerant's initialisation process and of course depending on the availability of a requested function in the card's hardware.

The BeOS R5 supports the following hooks:

Initialisation:
INIT_ACCELERANT
CLONE_ACCELERANT
ACCELERANT_CLONE_INFO_SIZE
GET_ACCELERANT_CLONE_INFO
UNINIT_ACCELERANT
GET_ACCELERANT_DEVICE_INFO
ACCELERANT_RETRACE_SEMAPHORE

2018 update:
Haiku has added support for the DDC/EDID channel which is used to determine the specifications of a screen connected to the graphicscard. This information is (for instance) shown in the Haiku Screen Preferences app. The hook we are talking about is:

GET_EDID_INFO

Based on the info from the DDC channel the driver can also specify to the app_server which mode is the 'native' (or best) mode to use for the connected screen. Sometimes the preconfigured state of the graphicscard hardware at boottime (partly) determines this, especially when part of the hardware register programming info is missing as that limits the options for the driver. The new hook is:

GET_PREFERRED_DISPLAY_MODE

Also added by Haiku are two hooks to get and set the screen's brightness. This will (mostly) be used for controlling laptop panels brightness.

GET_BRIGHTNESS
SET_BRIGHTNESS

Mode configuration:
ACCELERANT_MODE_COUNT
GET_MODE_LIST
PROPOSE_DISPLAY_MODE
SET_DISPLAY_MODE
GET_DISPLAY_MODE
GET_FRAME_BUFFER_CONFIG
GET_PIXEL_CLOCK_LIMITS
MOVE_DISPLAY
SET_INDEXED_COLORS
GET_TIMING_CONSTRAINTS

Powersave functions:
DPMS_CAPABILITIES
DPMS_MODE
SET_DPMS_MODE

Cursor management:
The cursor management hooks are only exported when the accelerant and the card both support a hardware cursor.

SET_CURSOR_SHAPE
MOVE_CURSOR
SHOW_CURSOR

2018 update:
While the (older) BeOS hooks are still supported in Haiku, Haiku also has added support for full color cursors. When Haiku's app_server determines which type of cursor to use, it will first try to use the new full color hardware cursor.

If that isn't supported, it will try the old type hardware cursor of which the hooks are mentioned directly above. If that also fails, it will fallback to use a software emulated cursor inside the app_server itself. One hook was added to add full color support:

SET_CURSOR_BITMAP

Acceleration engine synchronisation:
ACCELERANT_ENGINE_COUNT
ACQUIRE_ENGINE
RELEASE_ENGINE
WAIT_ENGINE_IDLE
GET_SYNC_TOKEN
SYNC_TO_TOKEN

2D acceleration:
The first four 2D hooks are used by the app_server. All hooks can be used by applications via BwindowScreen for example.
The 2D hooks are requested (again) by the app_server after every single modeswitch, also applications need to do this when they use the hooks directly. This way, depending on the engine's architecture, a different function can be used per mode to execute the acceleration function. For instance a quite different setup may be needed per different colordepth.
Also it's possible to *block* a function by not returning a hook in some mode(s) at all. In such a case it's required that the app_server or application does the work by itself (so in software). An 'everyday' example for this happening is when a virtualscreen is setup, where it's size is too big to still execute blits for example because the 'line length' (left to right in memory) is too big to specify to the hardware. So the acceleration engine simply does not support the mode, while the mode itself *is* supported. This can happen, because the memory reading hardware for the CRTC/DAC hardware is a different hardware block from the acceleration engine in the GPU.

SCREEN_TO_SCREEN_BLIT
FILL_RECTANGLE
INVERT_RECTANGLE
FILL_SPAN
SCREEN_TO_SCREEN_TRANSPARENT_BLIT
SCREEN_TO_SCREEN_SCALED_FILTERED_BLIT

Hardware overlay:
For the hardware overlay functionality the following hooks are used. Depending on engine architecture different functions may be needed for different colordepths. Also the overlay engine may not support all possible modes that can be setup in the card. Therefore the overlay hooks below are requested after every single modeswitch!
It is allowed to export all hooks, or no hooks at all: nothing in between. This way the accelerant lets the user know the current set mode on this card (or head) supports overlay or not.

OVERLAY_COUNT
OVERLAY_SUPPORTED_SPACES
OVERLAY_SUPPORTED_FEATURES
ALLOCATE_OVERLAY_BUFFER
RELEASE_OVERLAY_BUFFER
GET_OVERLAY_CONSTRAINTS
ALLOCATE_OVERLAY
RELEASE_OVERLAY
CONFIGURE_OVERLAY

The most important hooks are discussed one by one in the following paragraphs.

4.1.1 INIT_ACCELERANT

This hook is called first, so before any other hook is called. It will ask the already loaded kerneldriver for it's shared_info struct and make a clone of it for it's own use. If that fails (because the kerneldriver does not belong to this accelerant) then it will abort and report the error. Upon success the graphicscard will be initialized. This means the card will be taken through a 'coldstart' or a 'warmstart' cycle. The location in graphics memory which is going to hold the visible framebuffer, and the location which is going to hold the (optional) hardware cursor bitmap are determined. Needed semaphores are created and all other variables are initialized.
The accelerant will setup a modelist which is supported by this particular graphicscard with the particular screen now connected. This list will be made accessible via the shared_info struct. The maximum possible modelist for the driver is integrated in the accelerant's code, and is validated internally using the user hook PROPOSE_DISPLAY_MODE. This way the resulting actual modelist for the current hardware setup is generated. Apart from the screen's specs, for instance the maximum DAC (or serial monitor link, LVDS) speed and the available size of graphics memory have a large influence on the actual supported modes.

The BeOS app_server requires at least one mode per resolution in the modelist in order to correctly activate it's screen preferences panel with resolutions to select from. If a mode is missing, the belonging 'standard' resolution will be greyed out. Multiple modes per resolution can exist: there can be different colordepths, refreshrates and screen timing setups (varying relative place and length of the horizontal and vertical synchronisation pulses for example).

The modelist can later-on be obtained from the accelerant using it's own hook for that so the list can be used as a reference. This is also they way the app_server obtains this list.

Remarks:
Two things need to be done in a different way when compared to what the R4 graphics driver kit advertises, when creating the supported display_mode list in the internal function create_mode_list():

- Implicitly the kit says that for some hardware a larger pitch (distance between two 'lines') is sometimes needed to be able to generate a mode. The author ‘says’ this needs to be specified using the 'virtual_width' item: This is wrong however. The so called ‘slopspace’ which is meant here should be specified in 'frame_buffer_config.bytes_per_row' instead. Only this way the app_server will correctly use the slopspace: in the other case a virtual screen is generated in which can be 'panned’!
- The way 'Propose_display_mode' is called is not entirely correct. While a margin is correctly offered to Propose_display_mode for the requested pixelclock, the calling code does not check afterwards if the mode can be made within the specified limits. Instead of using the returned mode when 'Propose_display_mode' does not return B_ERROR status, it should only be used if this function returns B_OK instead.

2018 update:
Haiku has a new and improved screen preferences app when compared to the BeOS. These days it doesn't grey out non-existing standard (4:3) resolutions anymore for instance, but it simply populates the app with the modes supported by the driver. Therefore these days for instance widescreen modes can be set with it. No need for a custom screen preferences panel for this anymore.

4.1.2 CLONE_ACCELERANT

In case the driver and accelerant are already in use when an(other) application for example wants to load the accelerant for itself, this will be done using the clone_accelerant hook. Because the card hardware was already initialized before not much has to be done here.

First the hook will open the kerneldriver (so for a second or later time). After this the hook will ask the kerneldriver for its shared_info which will be cloned. Then the already before created modelist will be cloned.

4.1.3 UNINIT_ACCELERANT

When the current accelerant is the original one this function will destroy the semaphores the accelerant uses, along with the cloned shared_info and the mode_list. If the current accelerant is a clone, only the cloned shared_info and mode_list will be destroyed after which the kerneldriver will be closed.

4.1.4 ACCELERANT_RETRACE_SEMAPHORE

This hook only returns the semaphore the kerneldriver created for the function of synchronizing code with the screen's vertical retrace, important to be able to show tearfree moving video for example. If software wants to sync, it needs to lock the semaphore. This will not succeed until the kerneldriver releases this semaphore because it's vertical retrace interrupt routine is executing. When the software gets the lock it can immediately release it again and execute the screen's memory update that needs to be synced.

4.1.5 ACCELERANT_MODE_COUNT and GET_MODE_LIST

Accelerant_mode_count returns the number of graphics modes the mode_list contains (which list was created during the initialisation phase of the ‘original’ accelerant by this same accelerant). The application (or for example the app_server) intending to retrieve the mode_list, should make sure it has the memory space needed to hold this list which is later on retreived by calling hook Get_mode_list.

Remark:
The list as exported is used by the BeOS Screen Prefs panel to ascertain if the resolutions and colordepths in this panel are supported by the graphics driver. If for a certain resolution and colordepth more than a single display_mode exists, these modes will have varying (modeline) timing and/or refreshrates.
Whenever using the screen preferences panel the user switches between refreshrates while the exact matching mode does not exist in the mode_list, this application will in an adaptive way (by interpolating) put together modes itself from the modes in the list directly 'above' and 'below' the requested refreshrate. This way support for a broad range of screens on the market is guaranteed.

4.1.6 PROPOSE_DISPLAY_MODE

Propose_display_mode determines if the display_mode given to it can be set by the driver. When this is not the case, the accelerant will adapt the mode to become the 'closest' mode that *is* possible to set. Aside from this the accelerant will compare the modified mode to the also given limits 'high' and 'low'.
By letting an application set custom limits with a certain mode this application can check if this mode can be set within the desired limits.
The (user) application should make correct use of this limit checking feature: When Propose_display_mode should not check for limits, the application can create a single display_mode and feed this same mode three times to the Propose_display_mode hook (using the three pointers for 'mode to check', 'limit low' and 'limit high'). Since the limits will now be adapted by the accelerant (indirectly) the just modified mode by the accelerant will always be within the limits it sees.

Remarks:
The BeOS API class BWindowScreen uses Propose_display_mode via its function WindowScreen.SetFrameBuffer() to determine if a requested virtual size can be made for a previously set display_mode.
Unfortunately the BeOS uses Propose_display_mode in a dangerous way here: The BeOS should come up with limits that allow for some variation in most items in the display mode *except* for the virtual size members *virtual_width* and *virtual_height*, because only those two items should be checked.
The BeOS does not use it that way: In the current situation Propose_display_mode will let it know it cannot make the mode within the requested limits if for instance the pixelclock deviates only 1kHz. Which is a deviation that often occurs because of graphicscard hardware restrictions (in the PLL systems).

Please note that the display_mode is already set at the time the BWindowScreen object is created. So the mode can be made per definition aside from the changes in virtual size, requested via the SetFrameBuffer() function afterwards.

It is important to recognize that during creation of the BWindowScreen object a Set_display_mode command is given to the accelerant. This command is supposed to set the mode if at all possible. Potential adaptations to the mode while setting can *not* be reported back to the calling program, therefore a small deviation in for example the set pixelclock is not known there either. While this small adaptation has no influence on the mode at all (1kHz compared to 50Mhz pixelclock is a deviation of 20 parts per million!), BWindowScreen.SetFrameBuffer() will be told via Propose_display_mode about this deviation causing it to fail and exit with an error...

As a temporary solution (workaround) to this problem the accelerant's way of executing the Propose_display_mode function can be changed to not report small deviations outside of the given limits back to the calling program, *except* in:
- virtual_width;
- virtual_height;
- timing.h_display;
- timing.v_display.

4.1.7 SET_DISPLAY_MODE

Set_display_mode should activate the mode given to it, unless it cannot be done 'in any way'. Small deviations should be accepted by changing the mode so it will work. This shows from the fact that this function should validate the mode internally by (in turn) calling Propose_display_mode, while also the Set_display_mode command only aborts upon hard faults during execution of Propose_display_mode. Here Propose_display_mode is *not* used to check for deviations, it now only checks for ‘hard’ faults.

Remark:
Unlike as is shown in the R4 graphics driver kit, Propose_display_mode can best be called with three pointers to the display_mode 'target' to in effect disable limit checking.
The illogical manner of using a single(!) display_mode called 'bounds' holding a copy of 'target' is *not* advisable.

4.1.8 GET_FRAME_BUFFER_CONFIG

This function returns the 'fbc' struct belonging to the currently active display_mode to the caller. This fbc struct contains information which actually belongs inside the display_mode itself, since it is indispensable to correctly handle a display_mode. The app_server therefore uses this information to be able to correctly use the set display_mode.

fbc.bytes_per_row tells you the number of bytes that a display line is long, being a single pixel in height. The difference between '(display_mode.virtual_width * bytes/pixel)' and 'fbc.bytes_per_row' is the *slopspace* the current mode needs to have to function on the current graphicscard hardware. Slopspace is the non-used space to the right of the screen ('scrap paper '). This is the difference between:
- the requested width in pixels for the screen, and
- the needed width to comply to the graphicscard hardware restrictions for this.

Remark:
Different parts of the graphicscard (chip) often have different restrictions:
- CRTC-timing (control of the screen) usually needs visible widths dividable by 8;
- CRTC-memory access (for fetching pixel information that is i.e. fed to the DACs) needs a specific pitch between the different horizontal lines depending on the colordepth, because the hardware functions with a fixed number of bytes per ‘fetch’ (as in: uses batches, for i.e. electronic design simplicity and/or speed reasons: the size of the batches (also) depends on the buswidth to/from graphics RAM). For instance Matrox Millenium cards need a pitch which is dividable by 128 bytes, while Matrox G100 and later cards just need 16 (in 8bit color this means 16 pixels, while in 32bit color this means just 4 pixels) [11, 12];

- The acceleration engine has its own limits which must be dealt with. Matrox G100 and newer cards work here with a 32 *pixels* width, so *independant* of the colorspace in use!

In order for the graphicscard hardware to be able to accomodate a lot of (custom) horizontal resolutions in graphicsmodes, slopspace is needed. Horizontal resolutions dividable by 8 (pixels) are doable most of the time!

4.1.9 GET_PIXEL_CLOCK_LIMITS

This function returns the minimal- and maximal possible pixelclock for the given display_mode. The BeOS Screen preferences panel for instance uses this function to setup the lower- and upper limits for its refreshrate slider. Also predefined refreshrates (i.e. in a list) can be greyed out or disabled if after calling this function it's determined they cannot be done.

In the graphicscard hardware the maximum pixelclock will in most cases be dependant upon the colordepth chosen. The maximum pixelclock depends on:
- the maximum speed the DAC can cope with (so in Mhz);
- the speed the graphics chip 'core' is set to (mostly set to a fixed speed: although sometimes the speed depends on the kind of graphics mode in use: i.e. 2D/3D use, 'standard' VGA or 'extended' modes, single or dualhead modes);
- the graphics RAM speed (clockspeed) in combination with it's memory bus width (in bits, i.e. 128, 256 or 512 bits wide).

In modes with low colordepth mostly the maximum DAC speed can be set, while for higher colordepths the bottleneck suddenly becomes the RAM speed. This means that the maximal possible refreshrate is lower using higher colordepth modes than it is with lower depth modes.
When going beyond the maximum speed (by 'overclocking') as specified by the manufacturer, overheating with destruction can occur as a result. As a intermediate result it's possible that the card remains functional, but at a definite lower max. spec, going lower on each spec violation that occurs.
When going beyond the 'maximum' speed (by relatively 'underclocking') you'll see the same symptoms as with overclocking but in this case it's mostly non-destructive for the card's hardware.

When the (currently set) maximum RAM-fetching spec is violated it's very well possible that you can clearly see it happening: there will be malfuncions visible. For example:
- a fixed, green color on the pixel-'spots' where data is missing;
- you will see the screen through 'white snow' which is moving across the screen;
- you will see repeating pixels to the right (as the screen is 'built' in time from left-to-right and top-to-bottom).

The card's RAM's data is not only fetched from it for displaying it onscreen. This RAM is also accessed by write actions from the computer (program) by DMA or CPU or GPU (accelerated) access. Furthermore if it's dynamic RAM (so non-static RAM) also refresh cycles are happening regularly.
This means that the max. possible fetching speed is reached a bit earlier than you might think when 'designing' your hardware setup for a certain mode.
Some card types have a caching mechanism which you need to program explicitly for a certain strategy for dealing with all these concurrent accesses in order to get the best from the card. If you neglect to program this (or the specs are simply missing..) you'll have more limited modes at your disposal to choose from than is otherwise possible.

Remarks:
A remark about the BeOS Screen preferences panel behaviour: This application shields the user from setting very high refreshrates. As long as a mode can be done beyond 90Hz the upper settable limit will be fixed at 90Hz. The driver's influence only becomes visible here when 90Hz nolonger can be done.
Oddly enough such a restriction does not exist at the low end. If the driver says 4Hz is doable, the panel will simply indicate this.
It's because of this kind of (possible) behaviour in apps interfacing with the driver that it's advised to set decent limits yourself within the driver. The lowest setting should be limited around some 48Hz, because too low refreshrates can destroy your user's (cheap) screen just as good as too high refreshrates can.

Luckily these days every screen has it's own sanity checks built-in, which makes it shut-down automatically when being sent signals outside of it's designed specs. Or, as a lot of screens do these days, a 'out of range' message is displayed on the screen instead of the mode sent to it by the computer.
(48Hz is being mentioned as lower limit value because if someone would want to display PAL television output then 50Hz refresh might come in handy..)

A second remark about the BeOS Screen preferences panel: This panel (in R5) contains an *error*:
This application should ask for the pixelclock limits for the *suggested* display_mode as is visible in the panel, so not for the limits of the currently *active* display_mode like it is now. That would make the result much more logical to deal with for the user..

4.1.10 MOVE_DISPLAY

Move_display is used for virtual screens. Such screens have a resolution beyond the resolution currently showing on the monitor. Using the move_display function it's possible to move the visible area in position around the complete virtual screen area (as a 'viewport' to it) by panning (horizontal movement) and scrolling (vertical movement).

Please keep in mind that on some hardware modifying the starting adress pointer to the visible part of the (virtual) screen requires synchronisation with the screen's vertical retrace event. These cards do not contain double buffered hardware registers which automatically copy their content to the really active register(s) all at once after updating *all* user accessible (so 'shadow') registers in a *specific* order is done.
In such a case the driver needs to wait for the start of the vertical retrace before updating the starting adress registers because otherwise visible distortions will happen when moving the viewport position.
Retrace synchronisation can be accomplished here by polling a (standard VGA spec) register in the card's hardware for this event.
Be carefull though: the framebuffer's starting adress (i.e.) also needs to be set during the set_display_mode command. At this time however the screen has been temporary shut-down because no mode is active (it's in the process of being modified) so there will also not be any retrace event! A timeout counter (in a universal routine for polling this event) is necessary therefore...

- Applications can use the Move_display function via BWindowScreen.MoveDisplayArea(), when using BWindowScreen.SetFrameBuffer() a virtual resolution being higher than the screen's resolution has been set. This last item can only be done when BWindowScreen.CanControlFramebuffer() returns TRUE. Whether this function returns TRUE or FALSE in turn depends on the graphicscard driver. Control of this item runs via a certain display_mode flag.

- Whenever a virtual screen is used for displaying the desktop, the move_display function is called internally directly by the accelerant from within it's cursor control functions. Whenever the cursor is moved beyond the visible area onscreen, it will cause the viewport to follow.

Remarks:
An incompleteness concerning move_display exists in the app_server. While theoretically both a hardware- and a software cursor are supported (selection is done by the driver whether or not exporting the accelerant functions for cursor control: this may differ *per display_mode*), software cursor support by the app_server is not completely existing.

Whenever a software cursor is active the functions inside the accelerant for the hardware cursor are not used. This is not possible since these functions don't exist at that time outside of the accelerant itself. Inside the accelerant these functions are not useable at this time either.

This means that the responsibility of calling the move_display function in the accelerant now nolonger falls under the hardware cursor control in the accelerant itself, but under the software cursor control within the app_server. In the current BeOS version (R5) setting a virtual screen with software cursor works ok: although the viewport does not follow the cursor unfortunately.

A second limitation with respect to this accelerant hook is this: Because the lowest few bits of the framebuffer's starting adres often cannot be set, a minimum 'stepping granularity' exists for horizontal movement of the virtualscreen's viewport. This granularity might differ per hardware, per mode (colordepth), per head even.

Unfortunately the BeOS does not offer a function for applications to get information about this granularity in the API. Also there's no accelerant hook function for this. As a result applications need to execute a trick in order to more or less make 'sure' the requested startadres can be set. When an exact adres is important, an application should keep the lowest three or four bits 'reset' (zero) at all times. The larger the number of reset low-order bits, the higher the chance that the adress will be set exactly as requested.

While the BeOS R4 Graphics Driver Kit *does* speak about an accelerant hook called B_GET_MOVE_DISPLAY_CONSTRAINTS which would be implemented starting with R4.1, this never happened.

4.1.11 SET_INDEXED_COLORS

This function is used to set the hardware color palette in the graphicscard that is used for the 8-bit colorspace mode. The hardware color palette contains 3x256 bytes palette RAM, which serves as a lookup table to fetch a RGB color (24-bits) belonging to the value which is 'noted' as being pixeldata inside the framebuffer. So the pixels inside the framebuffer are *not* sent directly to the DAC, but instead serve as an index into the palette RAM. The values for R, G and B found inside that RAM at the given offset will be sent to the DAC instead. This way 256 colors can be selected from a 24-bit color 'palette' for simultaneous display on the screen. The pixeldata in the framebuffer serves as an index into all three palette RAMs: de same index applies for the R, G and B RAM.

In higher colordepths the palette RAM is unused as the pixeldata is sent directly to the DAC now (this is called a 'direct mode'). On some hardware the palette RAM is used for GAMMA (luminance) correction instead in these higher colordepth modes. GAMMA correction is used here to (slightly) adapt one or more of the base colors red, green and blue to correct for deviations in screen visualisation. Here the pixeldata is used again as index into the palette RAM (so 'indirect' mode). This time however the pixeldata for the basecolors R, G and B form *seperate* indexes in their corresponding color's palette RAM. The R, G and B values retreived from the palette RAM are now sent to the DAC.

The BeOS supports indexed mode only for 8-bit colordepth. The app_server determines which colors will be used in this mode and sets them using Set_Indexed_Colors. API class BScreen only offers functions to retrieve this mapping. So it's not possible for an application to change it.

At least one important reason why app_server should select the colors which will be used onscreen, is the possible use of hardware (video) overlay. App_server should keep a single exact color inside the palette which is *not* used onscreen, *except* during use of hardware overlay: in this specific case this color is used as a 'key' for the card's hardware to switch between displaying hardware 'accelerated' video or i.e. menu's or other overlapping windows ('color keying'). In order for the card's hardware to know when to switch, app_server sends this colorkey to both the application generating the video and the accelerant (by use of function 'configure_overlay' for the latter).

So app_server generates the colors that should be used for filling the palette RAM in 8-bit colordepth and sends it to the accelerant directly after such a mode is set via the set_indexed_colors hook. For all other colordepths the graphics driver needs to determine by itself which colors should be placed inside the palette RAM when indirect modes are being used. Therefore the palette RAM should best be shielded from the set_indexed_colors function and so as well for the app_server when these colordepths are used.

In theory the graphicsdriver is free to offer GAMMA correction to it's user: though the settings for such a thing are not supported by the BeOS. An alternative solution could be used though, such as using a driver settings file or so.

The downside of gamma correction use would be that the darkest and lightest colors cannot always be shown (as they would be 'clipping' at lowest or highest intensity) as usually there are only 256 color options for the 256 palette locations inside the palette RAM.

Please note that for instance Matrox Parhelia cards have 10 bit DACs so this range of cards probably also has 3x 10-bit wide palette RAM with 1024 palette locations, but it (probably) still 'suffers' from the same type of restriction. In practice this restriction is probably of no (real) importance.

On the other hand these days some screens have such a GAMMA correction option inside their own hardware already. This seems to be a more logical place for such a correction as well.

2019 update:
An extension to Haiku would be thinkable adding gamma correction as a extra setting for the screen preferences panel (for colordepths other than 8-bit). Such an update would require an API function to fetch and set a GAMMA palette from/to the app_server. App_server would be able to send this table to the driver using the existing set_indexed_colors hook. The graphicsdriver needs to 'export' the ability to accept a GAMMA table as a flag belonging to the currently set graphics_mode, and of course it should also accept the table into it's palette RAM if the current mode it has set supports the GAMMA correction. Because it's thinkable that even 8-bit mode would support GAMMA correction in theory (I haven't seen that yet though), maybe it would even be better to let the accelerant export a seperate hook especially to interface to the palette RAM for the purpose of executing the GAMMA correction.

Remarks:
Some cards have a palette RAM with 3x 6-bits depth as was specified in the original VGA standard, as opposed to 3x 8-bits depth as is used nowadays. Indexing of 8-bit color mode means in this case that the 6 most significant bits should be stored in the palette RAM: the BeOS does not account for the loss of the two least significant bits. For normal use this is no problem however.
When colorkeying is in use for hardware overlay it's another story. The key as given to the accelerant by the app_server via the configure_overlay hook should now be adapted: The lowest two bits of each base-color (R, G and B) needs to be reset because we have to be sure a 'match' will occur!
For modes with 15 (3x 5-bits depth) and 16 bits (R and B are 5 bits in depth, G 6 bits) in indirect mode there is no limitation as this bit-depth totally fits inside the palette RAM. 24-bit and 32-bit depth modes will not use the palette RAM (direct modes) so there's no problem here either.

The MagicGraph and MagicMedia graphics cards by manufacturer Neomagic[13] (which are used in laptops) for example have a 3x 6-bit wide palette RAM. Because these bits are located at the 6 least significant bitpositions on the used 8-bit bus, the colordata for input to the palette RAM should be 'shifted down' 2 bits te get the correct effect.

The Matrox G100-G550 cards for example have a 3x 8-bit palette RAM at their disposal.

4.1.12 GET_TIMING_CONSTRAINTS

This function is used to indicate the restrictions that the CRTC timing must meet. Here you can see, among other things, by which factor the horizontal timing must be divisible for valid display_modes (usually 8 'pixels'). The vertical timing usually works 'per pixel'.

4.1.13 SET_CURSOR_SHAPE

This function is used to write the bitmap data which is used to represent the cursor (mouse pointer) into the graphics card. Some (older) cards have a seperate and dedicated memory for this, while newer cards usually place the cursor bitmap in general graphics memory.

The cursor bitmap is overlayed on top of the screen image in more or less the same way as playing back video using hardware overlay does. The cursor functionality is a piece of dedicated hardware inside the graphics card, apart from (one or more) video overlay engines, and apart from showing graphics memory (framebuffer) on a screen.

Because changes to the shape of the bitmap can occur at any position on the screen and at any time (R5's BeIDE is notorious: the bitmap is overwritten with every single mouse move action!) it is important with some graphics cards to overwrite the bitmap only outside of the *moment* the cursor is being drawn onscreen. In order to do this without

too much delay or CPU time spent idling (i.e. while waiting for a vertical retrace), on most cards it's possible to ask a specific register which screen 'line' is currently in the process of being sent to the screen. If this line corresponds with a vertical position (viewport position, not a screenmemory position) on which the cursor bitmap is drawn (somewhere on the relatively fast drawn X-axis so to speak), visible distortions might occur if the bitmap is overwritten at that point in time.

Sometimes it can be handy to just shut-off the cursor temporary while updating its bitmap. Because the time needed to update it is very short, the resulting flickering is often not noticeable.

If the DAC and the CRTC functional blocks are placed in two seperate chips, it can become an even greater challenge: the CRTC timing signals will be your reference for the place of the cursor. A small mis-adjustment in the CRTC setup can simply make your cursor disappear because some sync puls just isn't 'seen'...

Luckily on newer cards there is often a hardware provision present which solves cursor disruptions itself automatically. This is done using double buffering techniques, for example with automatic update to the front buffer at the point in time where the last (shift) register for a coordinate has been updated. Otherwise it's sometimes very tricky to get the hardware cursor functioning without distortions as you probably gathered already.

Changing the cursor bitmap is used for example in a webbrowser to indicate that the cursor is hovering above a hyperlink, or in a PDF reader to indicate that the paper is being grabbed for moving when you press and hold a mouse button.

2019 update:
While the (older) BeOS hooks are still supported in Haiku, Haiku also has added support for full color cursors. When Haiku's app_server determines which type of cursor to use, it will first try to use the new full color hardware cursor. If that isn't supported, it will try the old type hardware cursor. If that also fails, it will fallback to use a software emulated cursor inside the app_server itself. One hook was added to add full color support:

4.1.14 Haiku only: SET_CURSOR_BITMAP

Set_Cursor_Bitmap is used to write a cursor bitmap into the graphics card in much the same way as with old hook Set_Cursor_Shape. The old hook only supports a 2-bit 'depth' type bitmap which gives only sort of a black and white type of cursor, with 'full transparancy' included by one direct value in this range of 4 possible 'colors'. The other 3 colors are 'black', 'white' and 'invert'.

The new hook offers a full-color cursor including alpha channel for transparancy (so 4x 8-bit for R, G, B and A). This new hook should only be exported by the accelerant for cards and modes which support this cursor in it's hardware. Newer cards mostly support this feature, like for example more modern nVidia graphics cards. Older cards mostly only support the old type hardware cursor, if any at all: No hardware cursor on the secondary heads for Matrox G400-G550 dualhead cards, and only the old type hardware cursor on the primary heads for example.

Of course transparancy is used because mostly we don't want to show a pure square bitmap on top of our screen's content, but for instance a 'hand' shape..

4.1.15 MOVE_CURSOR

Move_cursor is responsible for moving the cursor to the given coördinates on the screen. Because with virtual screens the space in which the cursor can move is larger than just the visible viewport to that screen, the possibility of the cursor leaving the visible part of the screen must be taken into account. If the cursor indeed moves beyond the viewport, move_cursor should call move_display internally. If hardware overlay functionality exists in the driver, also the configure_overlay hook should be called internally in this case, with the B_OVERLAY_TRANFER_CHANNEL flag explicitly cleared: Since only the position of the overlay output window moves and there is no input bitmap switch happening here.

The reason for the need to call move_overlay is that the backend scaler does it's internal calculations referenced to the starting adress as set in the CRTC, which is *not* the starting adress of the virtual screen (unless by chance we have the viewport currently located at the utter left-top position of the virtual screen of course). Since move_display changes the CRTC 'reference point', we need to update the overlay output position as well.

Also the calculation of the cursor position on the screen is done relative to the CRTC which is why we send the cursor hardware coördinates relative to the CRTC as well, so *not* the (direct) coördinates handed to us by the app_server since these are given relative to the virtual screen starting point.

Since furthermore changing the cursor position is not synchronized to sending the screen's data to the monitor, it is important on some graphic card types to update the cursor position registers during vertical retrace only. This depends upon double buffering being implemented or not in the position hardware registers on the graphics card. If not and the retrace is not being waited for, the cursor might start to 'jump' across the screen, because it temporary got an incorrect coördinate: Coördinates are often spread across multiple registers, which is why you cannot update them in an atomic fashion from the perspective of the hardware.

4.1.16 Haiku only: GET_EDID_INFO

Haiku has added support for the DDC/EDID channel which is used to determine the specifications of a screen connected to the graphicscard. This information is (for instance) shown in the Haiku Screen Preferences application.

This DDC communication channel sits inside the connector and cable to your screen in for example VGA, HDMI and displayport connections. It uses the I2C (Inter-Integrated-Circuit) protocol once developed by Philips, sometimes also living by the name I2S, to exchange digital information about the connected device(s).

This protocol is capable of adressing multiple devices: You'll find it also on your mainboard, on your graphicscard, on your video capture card etc. to communicate with all kinds of devices like temperature sensors, rotation speed sensors and possibly controllers, voltage detection sensors, TVencoders, TVreceivers, LVDS conversion encoders etc.

For our use with this hook we only look at the I2C channel called DDC on the connection to your screen, to retreive the EDID block of information that contains a list of valid display modes for the currently connected device (screen). This list is important for (among others) helping to prevent users from being confronted with a 'black screen' because Haiku or the graphics driver (did) set(s) a incompatible graphics mode for this screen. Having this list in the graphicsdriver is also very helpfull when implementing the Haiku only hook Get_preferred_display_mode, which hook is described below.

Please note that Haiku has library functions that your graphicsdriver can use to be able to use the DDC protocol and retreive and interpret the EDID information block. Saves a lot of work..

4.1.17 Haiku only: GET_PREFERRED_DISPLAY_MODE

Based on the info from the DDC channel the graphics driver can also specify to the app_server which mode is the 'native' (or best) mode to use for the connected screen. Sometimes the preconfigured state of the graphicscard hardware at boottime (partly) determines this, especially when part of the hardware register programming info is missing as that limits the options for the driver (as is the case with our nVidia graphics driver when used with non-VGA connected screens).

This is an important hook to implement since it can prevent the user from being confronted with a 'black screen' because i.e. the mode Haiku could otherwise start with is not supported by your currently connected screen. If this hook isn't implemented (correctly) for example trouble might arise when the user connects another screen (between boots), or when Haiku lives on a USB stick and the user takes this stick to another computer to start from it. Since that computer is probably made up of different hardware, it's very important to know the best (max) resolution that can be used in the current situation. If this resolution is below the one used last, this new best resolution will be used by Haiku instead.

4.1.18 The 2D acceleration functions

Apart from the six actual acceleration functions defined in BeOS R5 also synchronisation functions are implemented to support those. With these synchronisation functions it can be determined how many acceleration engines a card has, and a token can be aquired for an engine. This token has to be sent together with the drawing requests to the actual acceleration functions later on to let the right engine perform the actions.

The six acceleration functions are called with lists of items which should be processed/executed. The names of these functions are:

- SCREEN_TO_SCREEN_BLIT: Copy a rectangle from one place in the on(virtual)screen framebuffer to another place in this memory. This is the function which will give you the biggest acceleration effect of all. Whenever a window is being dragged, or inside a window scrolling or panning is done, this function will be used. If for instance you scroll down one text line in a text editor while the cursor was already on the last visible line, the whole content of the window apart from the top line will be copied one line upwards by the engine. This way the upper line disappears automatically, and only the new last visible line has to be drawn into graphics RAM manually by software. (So) source and destination rectangles may overlap.
- FILL_RECTANGLE: Fill a rectangle with a single color. This function is used to set the background color of the workspace for example, after which icons and other content is being drawn on top later on. Also windows being drawn have a certain background color which will be drawn using this function.
- INVERT_RECTANGLE: With this function a rectangle can be redrawn with the complement of its old colors. With this function is is rather simple to show a blinking selection for example. So this function reads the rectangle from the framebuffer and writes it back to the exact same spot internally using the complement color for each pixel within. If it does that twice you end up with the original content of course..
- FILL_SPAN: This function draws a horizontal line: So two X-coordinates and one Y-coordinate is given per line. A ‘span’ can be regarded as a rectangle with a single pixel in height.
- SCREEN_TO_SCREEN_TRANSPARENT_BLIT: This function is not used by the BeOS app_server. A rectangle can be copied in a transparant manner inside the on(virtual)screen framebuffer.
- SCREEN_TO_SCREEN_SCALED_FILTERED_BLIT: This function isn't used by the BeOS app_server either. With it you can copy a rectangle to another position inside the on(virtual)screen framebuffer. While copying, the source rectange is being scaled and filtered to prevent/minimize distortion patterns. Without the filtering function: pixels will be copied (scaling up) or dropped (scaling down), while using filtering (as this function always does) interpolation will be done instead. With this function the source and destination rectangles may *not* overlap. With the other functions overlap *is* allowed. This function can for instance be used for displaying moving video if hardware overlay is not available. In order to do it you'd need to create a virtual screen using BWindowScreen. This way the source rectangle(s) can be kept outside of the visible part of the virtual screen (so outside of the viewport area) while the destination is exactly the viewport area. You would not be scrolling and panning in this use of a virtual screen. Note: the nVidia driver Haiku has supports this function.

In theory some (most modern) acceleration engines can execute much more (variations of) functions than the BeOS uses, and on top of that also execute 3D functions. De SCREEN_TO_SCREEN_SCALED_FILTERED_BLIT function for example already uses a 3D acceleration engine function. In order to execute this function a so called 'texture engine' is needed: it's 3D properties are disabled so it does the 2D function we want done here.

Graphics cards for laptops and integrated graphics cards inside mainboard chipsets often (if older types, i.e. Neomagic's) don't support 3D acceleration. The acceleration engine 'manual' (if you are so lucky to be able to obtain it) usually shows exactly the functions the BeOS supports in it's acceleration engine interface, with the exception of SCREEN_TO_SCREEN_SCALED_FILTERED_BLIT: this function cannot be done there.
An(other) example for this are the Chips & Technologies chipsets for laptops like the CT69000[14]. Just fine chipsets for doing 2D acceleration, hardware overlay and hardcursor functions, but no 3D can be done.

Please note that you can decide per single acceleration drawing function (so: of the six mentioned above) if it will be integrated in the accelerant. Missing functions will be done in software ('fallback') by app_server or application software where needed.

Please also note that all acceleration functions work *inside* the (virtual) screen only, so not on pure 'offscreen' graphics memory.

2019 update:
While at some point in the (now distant) past Haiku's app_server used the driver's accelerated drawing functions, they were later on nolonger used. These days all drawing is done by app_server in software instead (so actually by the 'fallbacks'). The probable reasons for this decision are that these days the old bottleneck for datatransfer between main system memory and CPU to the graphicscard memory were lifted with the introduction of PCI-express (after AGP which still had that bottleneck for CPU transfers), the CPU's are much faster now, and the acceleration interface is (very much) outdated when compared to current hardware engines.

The downside is that on old systems you will notice relative slow responsiveness of Haiku when compared to the BeOS. Maybe an upside is that we would want to do accelerated desktop drawing using the 3D acceleration engines now, since these are much more available, hardware wise. Then again, since getting the needed specs from manufacturers is often not possible, it might not happen at all. Just doing old style 2D acceleration is much easier to reverse engineer than doing this for the much more complex 3D acceleration..

Example: accelerated video on a secondary graphics card.

In theory you could use a secondary (etc.) graphicscard in the system (so which is not used by the app_server), then interface with it directly via it's accelerant, to set up accelerated video. You could do this very nice in combination with a video producer/consumer node. In the years I worked on Matrox and nVidia TVout (2001-2003), Kevin Patterson wrote a BeOS media player that's a video producer node: and he also created a video consumer node.

The secondary graphics card would need to be 'coldstarted' by it's driver since only the primary graphicscard is inited by the system BIOS. At least the Matrox G100-G550 driver and the nVidia driver both support coldstarting (for VGA connected screens).
Now you could set a graphicsmode on this card, and see if it supports hardware overlay. If so, you could playback video using that. If not, you could see if it supports hook *SCREEN_TO_SCREEN_SCALED_FILTERED_BLIT*. If it does, you could use it for video playback (the nVidia driver has this function indeed, and it tested OK as well).
If this function doesn't exist either, you'd need to fallback to software rendering, or block play-back. You could grab say 4 (2x2), or 9 (3x3) graphicscards, and create a video wall with them.

'Coldstarting' a graphicscard using the driver is very nice to have, but it's just one of those things that requires lots of work, and lots of specs from the manufacturer. Therefore it would be much more practical if Haiku would be able to coldstart all non-primary graphicscards by use of their 'onboard' BIOSes. It would need to emulate a x86 realmode computer in order to pull that off since those BIOSes are written for that mode of operation. But then again, these days Haiku already posesses this functionality since it can switch graphicscard modes on the fly via VBE2 - vesa mode. This is also a realmode type of BIOS function..

4.1.19 The hardware overlay functions

In BeOS R5 en later zijn 9 functies voor overlay gebruik gedefinieerd. De hieronder eerstgenoemde 3 functies worden helaas niet gebruikt voor zover bekend. De eerste twee worden wel aangevraagd door de app_server maar nooit werkelijk gebruikt, de derde wordt zelfs niet aangevraagd. API toegang tot deze functies is voor zover bekend ook niet beschikbaar.

Omdat in R5 hardware overlay voor het eerst 'experimenteel' wordt ondersteund is het wel logisch dat de API nog niet af is, maar jammer is het wel. Momenteel moeten hierdoor applicaties een lijst met colorspaces afgaan en proberen ervoor een overlay-bitmap te definieren om te zien of hardware overlay wordt ondersteund: via trial and error dus.
Wanneer de API af zou zijn dan zou via de eerste drie hieronder genoemde accelerant overlay functies heel mooi kunnen worden bepaald of de driver overlay ondersteunt, en met welke colorspaces en features.

De gedefinieerde overlay functies zijn:
- OVERLAY_COUNT: geeft het aantal overlay units aan dat de kaarthardware bevat voor zover bekend. Op bestaande Be drivers is dit getal altijd één als overlay geimplementeerd is.
- OVERLAY_SUPPORTED_SPACES: geeft de namen van de ondersteunde input-bitmap colorspaces aandat ondersteund is via een pointer naar lijst, eindigend op B_NO_COLOR_SPACE. De meest ondersteunde colorspace in BeOS drivers is B_YCbCr422.
- OVERLAY_SUPPORTED_FEATURES: geeft met flags aan welke features worden ondersteund voor de aangegeven ondersteunde input-bitmap colorspace. Kandidaten zijn horizontale en verticale filtering (interpoleren tijdens verschalen dus), gespiegelde weergave en methoden voor colorkeying.
- ALLOCATE_OVERLAY_BUFFER: deze functie wordt gebruikt om ruimte voor een input-bitmap in het videogeheugen te reserveren via de constructie van een BBitmap met de B_BITMAP_WILL_OVERLAY flag gezet. Van de accelerant wordt verwacht dat hij zelf aan geheugenbeheer doet hiervoor, en bepaalt hoe groot de benodigde ruimte is.
- RELEASE_OVERLAY_BUFFER: hiermee wordt de reservering van de aangegeven input-bitmap in het videogeheugen gewist: het geheugen wordt dus vrijgegeven voor ander gebruik.
- GET_OVERLAY_CONSTRAINTS: deze functie geeft voor de aangegeven bitmap aan wat de minimale en maximale verschaal factor is, en wat de minimale en maximale output window grootte is. De API kan deze functie aanroepen via BBitmap.GetOverlayRestictions().
- ALLOCATE_OVERLAY: Als de overlay unit nog vrij is, wordt een token uitgegeven waarmee de applicatie die deze token ontvangt kan bewijzen dat hij de eigenaar van de overlay unit is. Als de unit al bezet is, wordt de applicatie geacht terug te vallen op software output. In principe kan meer dan een enkele overlay unit worden ondersteund via deze functie.
- RELEASE_OVERLAY: wordt gebruikt om de overlay unit weer vrij te geven. Tevens schakelt deze functie de overlay unit uit zodat de normale output weer op het scherm wordt weergegeven. Als een applicatie crasht, zorgt de app_server voor nette afhandeling van de lopende zaken.
- CONFIGURE_OVERLAY: zorgt voor de afhandeling van de aansturing van de overlay unit zelf. De inputbitmap welke moet worden weergegeven wordt aangegeven, dat deel ervan dat daadwerkelijk wordt gebruikt (hardware zoom), en de plaats en grootte van het uitvoerwindow wordt aangegeven (info voor verschalen zit hier dus ook in). Ook het token dat de applicatie kreeg tijdens allocate_overlay wordt doorgegeven.

Als een modeswitch plaatsvindt terwijl de overlay unit in gebruik is zal de app_server, voordat deze modeswitch plaatsvindt, zelfstandig alle buffers van de applicatie vrijgeven via RELEASE_OVERLAY_BUFFER. Verder zal hij RELEASE_OVERLAY aanroepen om de overlay output uit te schakelen en de unit vrij te geven.

Hierna zal de eigenlijke modeswitch plaatsvinden, waarna de app_server automatisch opnieuw de overlay_hooks aanvraagt. De app_server zal opnieuw buffers aan proberen te maken voor de applicatie (die onderwijl gewoon doorloopt!) en de overlay unit daarna weer aan het werk zetten.

Voor applicaties betekent dit dat ze geen kopieën van de pointers naar de buffers mogen maken, omdat deze pointers kunnen wijzigen door een modeswitch. Het gevolg zou kunnen zijn dat de applicatie een verkeerd deel in het videogeheugen beschrijft: Waar voorheen de buffers stonden, zou nu wel eens een deel van de zichtbare framebuffer (de desktop) kunnen staan.
Ook moeten applicaties rekening houden met NULL pointers wanneer (tijdelijk) geen buffers voorhanden zijn. In het ergste geval is de nieuwe framebuffer zoveel groter (als een hogere resolutie en/of kleurdiepte is gekozen door de gebruiker), dat er niet meer voldoende ruimte over is voor de overlay buffers.
Van de applicatie wordt verwacht dat dan bijvoorbeeld wordt teruggeschakeld op softwarematige output.

Belangrijk is te weten dat de backend scaler zijn interne berekeningen uitvoert met als referentiepunt het startadres van de CRTC, dus niet het startadres van het virtuele scherm. Net zoals het geval is met de cursor hardware. Voor meer informatie over de consequenties hiervan zie de beschrijving van de accelerant functie MOVE_CURSOR.

4.2 Conclusion

Het grootste deel van de functionaliteit van de videodriver zit in de accelerant. Alleen de namen van de hooks geven dit eigenlijk al aan. Deze accelerant interface is opgeruimd, en ondersteunt de kaartfuncties waarvoor de interface ontworpen is goed.

Er is slechts één manco in de ondersteuning: het gebrek aan de B_GET_MOVE_DISPLAY_CONSTRAINTS functie voor virtueel schermgebruik. Wanneer deze functie alsnog zou worden geïmplementeerd, zou deze functie naast de geplande granulariteit voor het verplaatsen van het scherm ook de volgende items moeten aangeven:
- De granulariteit van de breedte van het scherm;
- De maximale breedte van het scherm;
- De granulariteit van de hoogte van het scherm;
- De maximale hoogte van het scherm.

Het totaal zou hiermee op zes items uitkomen waardoor een stuk beter met virtuele schermen zou kunnen worden gewerkt dan nu het geval is.

De hardware overlay interface die experimenteel in BeOS R5 aanwezig is ziet er goed uit. Zodra volledige ondersteuning van de beschikbare functies in de API aangebracht is, kan er prettig mee worden gewerkt.

Het zou wèl mooi zijn als BeOS naast de eigen definities van colorspaces ook de ‘industry standard’ FourCC colorspaces zou kennen: De BeOS lijst is zeker voor hardware overlay onvolledig. Sommige colorspaces met hoge compressie wèl ondersteund in sommige hardware kan in BeOS helaas nièt worden gebruikt. Daarnaast moeten multi-platform applicaties (zoals de mediaplayer VLC[15]) nu een colorspace-vertaalslag uitvoeren.

De BeOS colorspaces zijn 16bits constanten gedefinieerd in een 32bits woord zodat implementatie van de fourCC definities niet moeilijk is. Waarschijnlijk kunnen beide lijsten zelfs zonder problemen worden gecombineerd.

Verder kan nog worden gezegd dat de accelerant interface richting API verder moet worden uitgebreid om de groter wordende ‘standaard’ mogelijkheden van de nieuwe generaties videokaarten bij te houden.

5. Flags

Een flag is in principe een enkel bitje. Dit bitje kent twee toestanden: waar of onwaar. Met flags kan een commando worden gegeven, of een status worden aangegeven. De flags in BeOS worden doorgegeven via 32bits woorden zodat 32 flags kunnen worden doorgegeven.

Er zijn diverse typen flags voor videodrivers. Ze hebben alle gemeen dat ze vanuit de API bruikbaar zijn voor applicaties. Er kan een onderverdeling van typen flags worden gemaakt:
- er zijn flags die worden doorgegeven aan de accelerant;
- er zijn ook flags die niet aan de accelerant worden doorgegeven. Dit zijn flags die de app_server vertellen hoe hij met de accelerant om moet gaan.

Flags die wel aan de accelerant worden doorgegeven bestaan weer uit twee soorten:
- status flags geven alleen een toestand in de accelerant aan;
- commando flags worden gebruikt om een opdracht aan de accelerant door te geven. Daarnaast geven ze soms de status van de uitvoering van het commando aan.

Een aantal flags worden in een enkele richting doorgegeven (richting accelerant: commando, vanuit accelerant: status), terwijl andere in principe in beide richtingen worden doorgegeven (commando met de status van het commando, of *alleen* status informatie: het ‘commando’ wordt dan genegeerd).

Als laatste zijn er nog enkele bijzondere flaggen:
- commando flags voor de accelerant welke in de API zitten maar daarnaast ook door de app_server zelf worden gebruikt;
- status flags die in de accelerant zitten voor de API maar daarnaast ook het gedrag van een class of de app_server (rechtstreeks) beinvloeden.

Wanneer deze bijzondere situatie van kracht is, is dit bij de gedetailleerde beschrijving van de desbetreffende flag vermeld. In onderstaande tabel is dit *niet* aangegeven.

Overzicht van videodriver gerelateerde flaggen in BeOS

naam van de flag	**plaats in de API**	**C**	**S**	**P**	**A**
voor overlay gebruik:					
B_BITMAP_WILL_OVERLAY	BBitmap: constructie	+		+	
B_BITMAP_RESERVE_OVERLAY_CHANNEL	BBitmap: constructie	+		+	
B_OVERLAY_TRANSFER_CHANNEL	BView: SetViewOverlay()	+			+
B_OVERLAY_MIRROR	BView: SetViewOverlay()	+			+
B_OVERLAY_FILTER_HORIZONTAL	BView: SetViewOverlay()	+			+
B_OVERLAY_FILTER_VERTICAL	BView: SetViewOverlay()	+			+
voor mode setup:					
B_SUPPORTS_OVERLAYS	BScreen: struct display_mode.flags		+		+
B_HARDWARE_CURSOR	BScreen: struct display_mode.flags		+		+
B_IO_FB_NA	BScreen: struct display_mode.flags		+		+
B_PARALLEL_ACCESS	BScreen: struct display_mode.flags		+		+
B_8_BIT_DAC	BScreen: struct display_mode.flags		+		+
B_DPMS	BScreen: struct display_mode.flags		+		+
B_SCROLL	BScreen: struct display_mode.flags		+		+
B_BLANK_PEDESTAL	BScreen: struct display_mode.timing.flags	+	+		+
B_TIMING_INTERLACED	BScreen: struct display_mode.timing.flags	+	+		+
B_SYNC_ON_GREEN	BScreen: struct display_mode.timing.flags	+	+		+
B_POSITIVE_HSYNC	BScreen: struct display_mode.timing.flags	+	+		+
B_POSITIVE_VSYNC	BScreen: struct display_mode.timing.flags	+	+		+

Legenda
naam: De naam zoals gedefinieerd is in de BeOS header files
plaats: De naam van de class en functie
C: commando: van API naar app_server of accelerant
S: status: van accelerant naar API
P: de app_server is doel
A: de accelerant is bron of doel

5.1 Flags voor overlay gebruik

In deze paragraaf worden de flags besproken die rechtstreeks van belang zijn voor de werking van hardware overlay in BeOS. Deze flags worden alleen doorgegeven ‘richting’ accelerant: ze kunnen niet worden teruggelezen.

5.1.1 B_BITMAP_WILL_OVERLAY

Bij de constructie van een bitmap wordt deze flag meegegeven om ervoor te zorgen dat de desbetreffende bitmap in het geheugen van de videokaart wordt aangemaakt in plaats van in het normale werkgeheugen van de computer. Het zetten van deze flag zorgt ervoor dat het systeem niet zelf een bitmap aanmaakt, maar in plaats daarvan aan de videodriver’s accelerant vraagt om de bitmap aan te maken. Dit gebeurt dan via de Allocate_overlay_buffer hook. De accelerant bepaalt aan de hand van:
- het nog beschikbare geheugen in de videokaart;
- de opgegeven colorspace waarvoor de bitmap moet worden aangemaakt, en
- de grootte van de bitmap in pixels

of de bitmap kan worden gemaakt. Als de accelerant tot de conclusie komt dat de bitmap niet kan worden gemaakt zal de constructie van de bitmap dus falen.

Het opgeven van de B_BITMAP_WILL_OVERLAY flag zet automatisch ook de B_BITMAP_IS_OFFSCREEN flag. Een overlay bitmap is namelijk per definitie ‘offscreen’: buiten het gezichtsveld in het videokaart geheugen geplaatst.

5.1.2 B_BITMAP_RESERVE_OVERLAY_CHANNEL

Deze flag is bijzonder omdat hij niet wordt doorgegeven aan de accelerant, maar wordt gebruikt om het gedrag van de app_server ten opzichte van de accelerant te beïnvloeden.

Wanneer een applicatie gebruik wil maken van hardware overlay, zijn er in principe twee mogelijkheden:
- Men maakt de benodigde bitmaps aan, met reserveert het overlay kanaal (de hardware dus), en men geeft de bitmaps weer;
- Men reserveert het overlay kanaal, men maakt de benodigde bitmaps aan, en men geeft ze weer.

De eerste optie wordt uitgevoerd door de bitmaps aan te maken zonder gebruik van de B_BITMAP_RESERVE_OVERLAY_CHANNEL flag. Wanneer nu via BView.SetViewOverlay() één van deze bitmaps wordt getoond met de B_OVERLAY_TRANSFER_CHANNEL flag in de functieaanroep gezet, zal ‘onder de motorkap’ de app_server er automatisch voor zorgen dat het overlay kanaal wordt gereserveerd.

De tweede mogelijkheid bestaat uit het aanmaken van de eerste bitmap met de B_BITMAP_RESERVE_OVERLAY_CHANNEL gezet, terwijl de overige bitmaps zonder deze flag worden aangemaakt. Nu kunnen de bitmaps worden getoond via het aanroepen van BView.SetViewOverlay() met de B_OVERLAY_TRANSFER_CHANNEL flag gezet.

De eerste optie vraagt dus impliciet om de overlay hardware, terwijl de tweede optie dit expliciet doet. Omdat de tweede optie dus vroeger in het proces vraagt om eigendom van de hardware, zal de kans van slagen hier groter zijn dan via de andere optie wanneer meerdere applicaties tegelijkertijd proberen overlay weergave te starten.

Opmerking:
Wanneer expliciet om het overlay kanaal wordt gevraagd, mag dit alléén gebeuren bij het aanmaken van de eerste bitmap. Wanneer wordt geprobeerd om een latere bitmap op dezelfde wijze aan te maken zal dit falen omdat de overlay hardware reeds bezet is.

Als een videokaart meer dan één overlay unit heeft, kan het aanmaken van meerdere bitmaps met de B_BITMAP_RESERVE_OVERLAY_CHANNEL flag gezet dus slagen. Weergave zal in principe dan wèl via verschillende overlay units plaatsvinden.
Kaarten met meer dan een enkele overlay unit (backend scaler) zijn door de auteur nog niet gesignaleerd.

5.1.3 B_OVERLAY_TRANSFER_CHANNEL

Deze flag is bijzonder omdat hij niet wordt doorgegeven aan de accelerant, maar wordt gebruikt om het gedrag van de app_server ten opzichte van de accelerant te beïnvloeden.

B_OVERLAY_TRANSFER_CHANNEL wordt gebruikt bij het aanroepen van BView.SetViewOverlay() om aan te geven dat de overlay unit moet switchen naar de meegegeven nieuwe overlay bitmap.

Normaal wordt tijdens dubbelgebufferde overlay weergave door de applicatie elk frame geschakeld tussen de verschillende overlay bitmaps die van te voren aangemaakt zijn. Terwijl één buffer op het scherm wordt getoond, wordt een ander in de achtergrond ververst met nieuwe data. Bij enkel gebufferde overlay weergave is dit niet het geval: de applicatie zorgt voor verversing van de altijd weergegeven bitmap tijdens de verticale retrace van de monitor.

Dit betekent dat BView.SetViewOverlay() door de applicatie alleen wordt aangeroepen tijdens dubbelgebufferde weergave van video. Bij enkelgebufferde weergave gebeurt dit éénmalig bij de start van de weergave.

Bij alle bovengenoemde toepassingen wordt SetViewOverlay() aangeroepen met de B_OVERLAY_TRANSFER_CHANNEL flag gezet. De flag dankt zijn bestaansrecht dan ook alleen aan de interne werking van de app_server.

De app_server kan namelijk ook geheel zelfstandig de accelerant functie CONFIGURE_OVERLAY aanroepen die ook wordt gebruikt voor de uitvoer van de BView.SetViewOverlay() API functie. De reden hiervoor is dat het heel goed mogelijk is dat de gebruiker het uitvoervenster op het scherm verplaatst of verschaalt. In zo'n geval wordt de applicatie niet lastiggevallen om aanpassingen te doen, maar geeft de app_server de nieuwe gegevens zelfstandig aan de accelerant door.
Omdat de input bitmap nu niet hoeft te worden geschakeld (dat is de verantwoordelijkheid van de applicatie als de tijd rijp is), wordt de B_OVERLAY_TRANSFER_CHANNEL flag nu niet gebruikt.

Opmerking:
Er is nog een situatie waarin het aanroepen van de accelerant functie CONFIGURE_OVERLAY plaatsvindt met niet gebruikte B_OVERLAY_TRANSFER_CHANNEL flag: het betreft hier een interne aanroep vanuit de MOVE_CURSOR functie.
Wanneer in een virtueel scherm de cursor door beweging buiten het zichtbare deel ervan valt, wordt het zichtbare deel van het scherm verplaatst. Omdat de overlay unit zijn uitvoer aan het zichtbare scherm (het CRTC buffer startadres) relateert, dient ook de 'nieuwe uitvoerpositie' van het overlay uitvoervenster aan de hardware te worden doorgegeven. Ook nu hoeft niet te worden geschakeld naar een andere inputbitmap, dus is de B_OVERLAY_TRANSFER_CHANNEL 'gecleared'. Hierdoor kunnen de nodige berekeningen voor de overlay unit bovendien sneller worden uitgevoerd omdat maar een deel van het geheel behoeft te worden ververst.

5.1.4 B_OVERLAY_MIRROR

Deze flag kan door de accelerant worden gebruikt als trigger om de overlay output horizontaal te spiegelen wanneer de hardware dit ondersteunt. Deze functie kan handig zijn wanneer video wordt weergegeven via een projectiescherm met een spiegel in plaats van op een standaard computer monitor.
Uitvoer van dit commando in de accelerant is optioneel en het wordt waarschijnlijk (bijna) nooit gebruikt.

5.1.5 B_OVERLAY_FILTER_HORIZONTAL

Via deze flag kan een applicatie aan de accelerant vragen om de hardware te laten filteren (interpoleren) tijdens horizontaal schalen van de overlay uitvoer. Als de flag niet wordt gebruikt, wordt kopiëren van pixels toegepast bij omhoogschalen, en worden pixels 'gedropt' (laten vervallen) tijdens naar beneden schalen.
Het implementeren van deze flag in de accelerant is optioneel, maar wordt sterk aangeraden. De beeldkwaliteit neemt behoorlijk toe wanneer interpolatie wordt toegepast.

5.1.6 B_OVERLAY_FILTER_VERTICAL

Via deze flag kan een applicatie aan de accelerant vragen om de hardware te laten filteren (interpoleren) tijdens vertikaal schalen van de overlay uitvoer. Als de flag niet wordt gebruikt, wordt kopiëren van pixels toegepast bij omhoogschalen, en worden pixels 'gedropt' (laten vervallen) tijdens naar beneden schalen.
Het implementeren van deze flag in de accelerant is optioneel, maar wordt sterk aangeraden. De beeldkwaliteit neemt behoorlijk toe wanneer interpolatie wordt toegepast.

5.2 Flags voor mode setup: mode.flags

Mode.flags worden in principe door applicaties via de API doorgegeven aan de accelerant via BScreen.SetMode() of via BScreen.ProposeMode(). Via BScreen.GetMode() en BScreen.ProposeMode() kunnen ze ook weer worden opgevraagd. De app_server zelf maakt in principe ook gebruik van deze flags voor communicatie met de accelerant.

In de accelerant worden de modes en dus de flags 1:1 doorgegeven aan of van de SET_DISPLAY_MODE, PROPOSE_DISPLAY_MODE en GET_DISPLAY_MODE hooks.

5.2.1 B_SUPPORTS_OVERLAYS

Via deze statusflag geeft de accelerant aan of de huidige display_mode op de kaart hardware overlay ondersteunt. Op bijvoorbeeld Matrox kaarten wordt hardware overlay niet ondersteund in interlaced modes.

5.2.2 B_HARDWARE_CURSOR

Via deze statusflag geeft de accelerant aan of de huidige display_mode op de kaart een hardware cursor ondersteunt. Helaas luistert de app_server niet naar deze flag. Als dit wel zo was dan was het mogelijk om terug te vallen op een software cursor wanneer naar een display_mode zou worden geschakeld welke geen hardcursor ondersteunt.

Zoals het nu is (BeOS R5) kan deze keuze alleen worden gemaakt tijdens systeemstart via het wel of niet exporteren van de accelerant CURSOR hooks, aangezien deze hooks slechts éénmalig door de app_server worden aangevraagd. Wanneer deze hooks na elke modeswitch opnieuw door de app_server zouden worden aangevraagd, zou ‘on the fly’ schakelen tussen soft en hardcursor ook mogelijk zijn.

Omschakelen kan handig zijn wanneer een display_mode nèt niet kan worden gemaakt door gebrek aan kaartgeheugen: de hardware cursorbitmap neemt meestal een stukje van dit geheugen in beslag.

5.2.3 B_IO_FB_NA

Deze flag geeft aan dat het videogeheugen van de grafische kaart niet mag worden gebruikt door de app_server wanneer de acceleratie engine van de kaart ermee bezig zou kunnen zijn. Waarschijnlijk is deze situatie alleen bij oudere kaarten van toepassing.

B_IO_FB_NA en/of B_PARALLEL_ACCESS hebben waarschijnlijk invloed op de werking van de API class BDirectWindow. BDirectWindow kan in principe worden gebruikt in fullscreen mode en in windowed mode. In het BeBook is bij de functie BDirectWindow.SupportsWindowMode() aangegeven dat de beschikbaarheid van windowed mode afhangt van de grafische kaart hardware.
Genoemd als voorwaarden voor de bruikbaarheid van windowed mode zijn:
- Hardware cursor ondersteuning,
- DMA ondersteuning en
- Parallelle toegang tot de kaarthardware.

5.2.4 B_PARALLEL_ACCESS

B_PARALLEL_ACCESS geeft aan dat parallelle toegang tot de kaart is toegestaan in de huidige display mode. Waarschijnlijk alleen een probleem bij oudere kaarten.

B_IO_FB_NA en/of B_PARALLEL_ACCESS hebben waarschijnlijk invloed op de werking van de API class BDirectWindow. BDirectWindow kan in principe worden gebruikt in fullscreen mode en in windowed mode. In het BeBook is bij de functie BDirectWindow.SupportsWindowMode() aangegeven dat de beschikbaarheid van windowed mode afhangt van de grafische kaart hardware.
Genoemd als voorwaarden voor de bruikbaarheid van windowed mode zijn:
- Hardware cursor ondersteuning,
- DMA ondersteuning en
- Parallelle toegang tot de kaarthardware.

5.2.5 B_8_BIT_DAC

Deze statusflag geeft aan dat de DAC in 8-bit mode is. Dit is alweer een wat vage flag qua definitie. Waarschijnlijk moet deze flag alleen bij oudere kaarten 'gecleared' zijn.
Vastgesteld is in ieder geval dat de app_server deze flag niet gebruikt om onderscheid te maken tussen kaarten met 3x6-bit breed palette RAM en 3x8-bit breed palette RAM. Met èn zonder deze flag blijft de app_server alle 8-bits gebruiken voor het opstellen van het B_CMAP8 colorspace kleurenpalette.

5.2.6 B_DPMS

Deze flag geeft aan dat de huidige display_mode DPMS ondersteunt. (DPMS staat voor 'Display Power Management System', gebruikt voor energiebesparende standby modes van de monitor wanneer deze niet wordt gebruikt.)

5.2.7 B_SCROLL

B_SCROLL geeft volgens het BeBook aan dat een virtueel scherm wordt gebruikt: een groot scherm wordt gesimuleerd door het rondscrollen in een kleiner scherm.
Het lijkt er echter op (na gedane testen) dat deze flag gezet moet zijn wanneer de driver virtuele schermen ondersteunt in de huidige display_mode. Dit betekent dat de flag ook gezet moet zijn wanneer de huidige display_mode zelf (nog) geen virtueel scherm voorstelt!
Deze flag wordt *soms* doorgegeven aan de API (R5): BWindowScreen.CanControlFrameBuffer() geeft soms een FALSE status terug wanneer deze flag niet gezet is! Dit betekent dat BWindowScreen in dit geval niet kan worden gebruikt voor virtuele schermen: De lidfuncties SetFrameBuffer() en MoveDisplayArea() zullen dan niet werken.

Het is dus verstandig om deze flag altijd te zetten wanneer de driver 'volledig' is.

5.3 Flags voor mode setup: mode.timing.flags

Mode.timing.flags worden in principe door applicaties via de API doorgegeven aan de accelerant via BScreen.SetMode() of via BScreen.ProposeMode(). Via BScreen.GetMode() en BScreen.ProposeMode() kunnen ze ook weer worden opgevraagd.
In de accelerant worden de modes en dus de flags 1:1 doorgegeven aan of van de SET_DISPLAY_MODE, PROPOSE_DISPLAY_MODE en GET_DISPLAY_MODE hooks.

Timing.flags kunnen worden opgevat als commando- en status flags, naargelang de wensen van de driver.

5.3.1 B_BLANK_PEDESTAL

Deze flag geeft aan dat een '7.5 IRE blanking pedestal' moet worden gebruikt in plaats van een '0.0 IRE blanking pedestal'. Meestal wordt 0.0 IRE gebruikt, en meestal is dit gefixeerd in de driver. Waarschijnlijk is deze flag alleen nodig voor oudere monitoren.

Wanneer de flag gezet is, is het de bedoeling dat het videosignaal naar de monitor in spanningsnivo een beetje wordt opgetild ten opzichte van het synchronisatiesignaal nivo. Dit is waarschijnlijk een hulpmiddel voor oudere (sync_on_green) monitoren om betrouwbaar onderscheid tussen video- en timingssignalen te kunnen maken.

5.3.2 B_TIMING_INTERLACED

Als deze flag gezet is, wordt van de driver verwacht dat hij de huidige display_mode als interlaced mode opzet. Een interlaced mode is een mode waarbij een frame wordt opgesplitst in twee fields: een even- en een oneven field. Deze velden worden na elkaar op het scherm afgebeeld. Het gevolg is dat met een lage signaal bandbreedte toch een hoge resolutie kan worden weergegeven. Een field bevat alleen de even of de oneven horizontale 'scanlijnen' van een beeld.

Interlaced weergave wordt bijvoorbeeld nog steeds bij televisie toegepast, en bij oudere VGA monitoren. De meeste huidige videokaarten en monitoren ondersteunen geen interlaced modes meer: alleen progressive scan modes worden tegenwoordig toegepast. Uitzondering hierop zijn TVout modes: deze werken in principe in interlaced mode omdat een TV toestel ermee werkt.

5.3.3 B_SYNC_ON_GREEN

Sync_on_green geeft aan dat de synchronisatiesignalen voor de monitor gesuperponeerd moeten worden verstuurd op het groene videosignaal, in plaats van via separate signaallijnen. Oudere monitoren hebben soms geen aparte synchronisatie ingangen en hebben dit systeem dan dus nodig.
Voor zover bekend hebben sync_on_green monitoren geen DPMS ondersteuning. Het is waarschijnlijk zo dat wanneer sync_on_green voor een kaart geactiveerd is, DPMS niet meer wordt ondersteund. DMPS moet dan dus worden uitgeschakeld in de driver. Voor Matrox MGA kaarten geldt dit in ieder geval.

Sync_on_green wordt tegenwoordig eigenlijk niet meer gebruikt en is alleen in sommige oudere videokaarten ondersteund. Matrox kaarten na de G200 bijvoorbeeld missen de benodigde hardware, terwijl de oudere kaarten wél sync_on_green ondersteuning bieden.

5.3.4 B_POSITIVE_HSYNC en B_POSITIVE_VSYNC

Bij originele VGA monitoren en in originele VGA modes wordt de polariteit van de synchronisatie signalen gebruikt om aan te geven welke resolutie actief is. Tegenwoordig maakt het de monitoren niet meer uit welke polariteit ze aangeboden krijgen: ze zijn adaptief in alle timings-opzichten. Voor veel modes worden tegenwoordig positieve synchronisatie signalen gebruikt.

De modelist zoals deze wordt geëxporteerd door de accelerant bevat zowel modes met negatieve als positieve synchronisatie signalen. Deze lijst wordt meestal (ten dele) gekopieerd van de lijst zoals vermeld in de R4 Graphics Driver Kit.

5.4 Conclusie

Het bedoelde gebruik van de videodriver gerelateerde flags is in BeOS helaas niet optimaal gedocumenteerd. Dit is de reden dat in dit document niet met zekerheid de exacte functie van elke flag aangegeven is.

Het is verstandig om op dit gebied verder onderzoek te doen om zo goed mogelijk de ontbrekende informatie boven water te halen. Datgene wat niet te achterhalen valt moet opnieuw worden gespecificeerd zodat in de toekomst goed gebruik mogelijk is.

Een interessant verschijnsel bij de implementatie van de flags welke worden doorgegeven aan- of van de accelerant is dat ook de bitposities die geen definitie kennen gewoon worden doorgegeven. In theorie zouden deze namelijk kunnen worden gereset door de app_server om ongedefinieerde toestanden te voorkomen.

Omdat de niet gedefinieerde bitposities ongewijzigd worden doorgegeven, terwijl applicaties en drivers in de praktijk de niet gebruikte bitposities keurig ‘gereset’ aanmaken, is het mogelijk een eigen uitbreiding in de flags op te zetten. De Matrox driver maakt hier bijvoorbeeld dankbaar gebruik van om aan applicaties te laten weten of TVout en dualhead op een kaart worden ondersteund. Daarnaast kunnen met andere custom flags de respectievelijke modes worden geactiveerd.

Voor uitbreidingen van de API en de driverinterface bevinden zich in de flag-ruimte dus mogelijkheden die direkt inzetbaar zijn. Het zou prettig zijn als hierover zou worden nagedacht en er een ‘standaard’ voor opgezet zou worden. Als dit niet gebeurt bestaat bijvoorbeeld het gevaar dat voor elke driver ‘met uitbreidingen’ een eigen preferences applicatie moet worden geschreven.

6. Het schrijven van de driver

Bij het schrijven van een videodriver zijn een aantal zaken van belang:
- Er is een plan nodig dat aangeeft in welke volgorde de onderdelen kunnen worden opgebouwd;
- Er moeten mogelijkheden zijn voor het testen van de driver; en
- De driver moet zodanig worden opgebouwd, dat de stabiliteit ervan zo goed mogelijk is gewaarborgd.

In dit hoofdstuk wordt op deze onderwerpen ingegaan. De hier gegeven informatie vormt een belangrijk stuk gereedschap bij het daadwerkelijk bouwen van een videodriver.

6.1 Stappenplan

Het schrijven van de videodriver wordt in het nu volgende in logische, testbare stappen beschreven. Voordat er kan worden begonnen, moet echter een besluit ter voorbereiding worden genomen.

6.1.1 Voorbereidingen

Hoe wordt de driver getest? Hiervoor zijn twee methoden denkbaar.
- Met twee videokaarten: Het systeem kan worden voorzien van twee videokaarten waarvan één exemplaar wordt gebruikt om op te werken via een reeds bestaande driver, terwijl de andere wordt gebruikt om de nieuwe driver voor te bouwen en te testen, of:
- Met één videokaart: Er kan worden gewerkt met één kaart en de 'aanstaande' driver ervoor terwijl deze driver in ontwikkeling is.

Twee videokaarten.
In het eerste geval is een testprogramma nodig welke de nieuwe accelerant rechtstreeks in gaat laden, zodat de functies stuk voor stuk kunnen worden getest terwijl BeOS er zelf niet aankomt. Als de driver dan voldoende gevorderd is om er display_modes mee te kunnen instellen, kan de andere kaart uit het systeem worden verwijderd zodat BeOS de Desktop via de nieuwe driver en kaart kan laten zien. De laatste stappen van de driver ontwikkeling kunnen dan op dezelfde wijze worden gedaan als via onderstaande methode.

Om met twee kaarten te kunnen werken is een truc nodig: In het BIOS van het systeem moet de kaart waarvoor de driver gaat worden geschreven worden ingesteld als primaire kaart. Als dit niet mogelijk is (er zijn dan alleen PCI kaarten aanwezig), kan de keuze voor een primaire kaart worden geforceerd door te schuiven met de 'insteek'volgorde van de kaarten in de PCI sloten in het systeem.

De primaire kaart is de kaart waarop de systeemstart te zien is (deze wordt door het systeem via de kaartBIOS geïnitialiseerd): de secundaire kaart blijft uitgeschakeld.

Naast de truc is er een voorwaarde waaraan de bestaande driver voor de nu secundaire kaart moet voldoen: Deze driver moet de kaart 'koud' kunnen starten. Dit houdt in dat de kaart geheel moet worden geïnitialiseerd: onder andere moet de geheugensoort worden ingesteld samen met de refresh en het clocksignaal ervoor. Helaas voldoen weinig BeOS drivers aan deze voorwaarde. Voor zover bekend worden alleen de Matrox kaarten volledig koud gestart. Dit geldt zowel voor de Be drivers, als voor de openBeOS Matrox driver (G100-G550 kaarten).

Er is *nog* een voorwaarde: Ten minste één van de kaarten mag géén gebruik maken van ISA resources. In de ISA standaard is namelijk géén ruimte voor het plaatsen van twee kaarten. Wanneer dit toch gebeurt zullen conflicten optreden omdat beide kaarten op dezelfde I/O adressen worden geplaatst.

Eén videokaart.
In het laatste geval is het mogelijk om een driver te schrijven op een testbare wijze op een systeem waarin geen twee kaarten kunnen worden geïnstalleerd, zoals in een laptop het geval is. Erg prettig is het wel als minimaal VBE2[16]

ondersteuning voor de kaart bestaat. Als dit niet zo is moet alles tot en met de modesetup ‘blind’ worden geschreven omdat niet kan worden getest (loggen kan natuurlijk wel). Daarna kan bij succes de driver worden gebruikt terwijl hij verder wordt ontwikkeld.

Het proces van het schrijven van de driver wordt in dit document beschreven voor een kaart waarvoor VBE2 ondersteuning aanwezig is. Er wordt dus geen testprogramma gebruikt om de accelerant rechtstreeks in te laden. VBE kan de kaart op verzoek voor ons instellen op een van te voren bepaalde gefixeerde enhanced mode. Hierdoor wordt de kaart in feite op het nivo van stap 11 gebracht (ingeschakelde enhanced mode) van het stappenplan voor het schrijven van een videodriver dat hieronder wordt beschreven. Hieraan ontleent dit stappenplan ook zijn werking. Daarnaast verklaart dit waarom de VBE-mode instelling kan worden verwijderd zodra stap 11 met succes is afgesloten.

VBE
VBE ondersteuning bestaat op twee manieren. Meestal zit de ondersteuning volledig in het BIOS van de kaart. Bij oudere kaarten is dit niet altijd (geheel) zo en moet een DOS TSR worden geladen om support te verkrijgen. Als dit nodig is, moet BeOS via DOS worden geladen. Dit is standaard mogelijk met de personal edition versie van BeOS R5: R5PE[17]. Deze versie van BeOS wordt standaard in een enkele file geinstalleerd als een Windows98 (of Linux) applicatie. In deze file bevindt zich dan een volledige virtuele harddisk met daarop BeOS.
Door nu ‘kaal’ naar een DOS prompt te starten (dus zonder dat Windows98 actief is), kan zo’n DOS TSR worden geladen. Hierna kan worden doorgestart naar BeOS via de folder waarin BeOS is geïnstalleerd. Hiervoor moet dan de file loadbeos.com worden gestart.

Ook een versie van BeOS geinstalleerd in een eigen partitie kan op deze manier worden gestart. Hiervoor zijn alleen de files loadbeos.com (de loader) en zbeos (de kernel) in de DOS partitie nodig.

Een VBE TSR moet worden betrokken bij de fabrikant van de videokaart omdat deze TSR kaartafhankelijk is.

6.1.2 Stap 1: VBE2 (VESA mode) activeren

Via de applicatie VesaAccepted op BeBits[18], of rechtstreeks via de ‘vesa’ settings file in de directory /boot/home/config/settings/kernel/ is het mogelijk om een display mode in te stellen voor gebruik in BeOS zonder een videodriver voor een kaart tot de beschikking te hebben. BeOS zal bij de aanwezigheid van een geldige vesa settings file proberen de VBE2 hook van het videokaart BIOS aan te roepen voordat protected mode van de CPU wordt ingeschakeld (VBE2 is geschreven voor realmode gebruik zoals dat in DOS het geval is).
Nadat op deze manier de gevraagde mode (bijvoorbeeld 800x600 pixels in 8bit kleurdiepte) ingesteld is, wordt overgeschakeld naar protected mode en wordt BeOS geactiveerd. De display mode kan nu niet meer worden gewijzigd, daarvoor is een herstart van het systeem nodig: realmode programma’s werken niet in protected mode. Om de optimale snelheid met scrollen in sourcefiles en dergelijke te verkrijgen, wordt aanbevolen om in 8bit kleurdiepte te werken: 16bit kleurdiepte bijvoorbeeld betekent dat de CPU twee maal zoveel data moet verplaatsen voor beeld gerelateerde functies dan dat met 8bit het geval is.

Mocht bij het starten van de app_server (welke zorgt voor het laden van een videodriver en het weergeven van de desktop erop) nu toch een videodriver worden gevonden welke de aanwezige kaart ondersteunt, dan wordt deze driver geladen en gebruikt. Het activeren van de VBE mode was dan dus overbodig en is in feite ongedaan gemaakt. De VBE mode is dan door de driver overschreven.

Wanneer er géén driver wordt gevonden, dan is dus een VBE mode actief en hebben we een keurig kleurenscherm zonder acceleratie, hardcursor en overlay. Er kan nu dus op het systeem redelijk goed worden gewerkt.

Een overzicht van de startmethode van BeOS is opgenomen als plaatje in hoofdstuk 3. Hier valt goed te herkennen hoe bijvoorbeeld de VBE ondersteuning door BeOS wordt uitgevoerd.

Als een driver wordt geschreven voor een reeds ondersteunde kaart wordt als het goed is de driver straks in de add-ons directory structuur geïnstalleerd (zie Hoofdstuk 3.1: Interface naar het OS) zodat de nieuwe driver automatisch voorrang krijgt op de bestaande driver.

Als alternatief kan de bestaande driver worden verwijderd zodat deze zeker niet zal worden geladen. Nodig is dit niet tenzij deze driver problemen veroorzaakt. De nieuwe driver maakt straks automatisch gebruik van de iets eerder gestarte VBE2 mode.

6.1.3 Stap 2: Niet werkzame driver installeren

Nu wordt het tijd om een geschikte opensource kandidaat driver uit te zoeken die we gaan kopiëren en aanpassen voor de nieuwe, eigen driver. Selectie criteria zijn bijvoorbeeld het vinden van een driver met een programmeerstijl die bij de programmeur van de nieuwe driver past, en bijvoorbeeld of de ondersteunde kaart van de brondriver lijkt op de kaart waar de nieuwe driver voor gaat worden gebruikt qua specificaties en manier van aansturen. Vragen die daarbij kunnen worden gesteld zijn onder meer:

- Is ISA I/O space toegang wel of niet nodig vanuit de accelerant?
- Is PCI configuration space toegang nodig vanuit de accelerant?
- Ondersteunen de nieuwe- en de bron driver TVout en/of dualhead modes?

Wanneer een keuze voor een brondriver gemaakt is, kan de kopie worden gemaakt. In de code van de kopie accelerant wordt nu alle registeraansturing uitgeschakeld zodat alleen de high-level code operationeel blijft. Dit is de code die de interface naar de driverhooks uitvoert.
In de kopie kerneldriver wordt de herkenning en het mappen van de kaart bijgewerkt zodat de huidige kaart wordt ondersteund. Eventuele registerprogrammering anders dan bedoeld voor het mappen van de kaart wordt uitgeschakeld. Denk om het activeren van colormode bij de standaard VGA registers bijvoorbeeld: de ISA adressering hangt hier gedeeltelijk vanaf (bijvoorbeeld de CRTC index: op ISA I/O adres 0x03b4 in zwart/wit mode, 0x03d4 in colormode).

Nu kan de nieuwe driver worden geïnstalleerd. Als alles goed is gegaan, zal bij een reboot de nieuwe driver en accelerant worden geladen terwijl de VBE mode eronder actief blijft: Er wordt tenslotte niets overschreven.

Alleen de framebuffer toegang wordt nu daadwerkelijk door de driver en accelerant beschikbaar gesteld aan BeOS. BeOS zal de Desktop op het aangegeven adres binnen de framebuffer opbouwen. Als dit het RAM offset adres $0 is, werkt de schermuitvoer waarschijnlijk naar behoren, omdat de VBE mode waarschijnlijk hetzelfde adres gebruikt voor weergave op het scherm.

Het is in ieder geval in deze stap de bedoeling dat deze framebuffer toegang werkt en de VBE mode niet verstoort. Ook is het erg handig om log functies naar een file in te bouwen in de accelerant voor testdoeleinden. Zo kan bijvoorbeeld eenvoudig omonstotelijk worden bewezen dat de accelerant daadwerkelijk actief is: anders wordt er tenslotte niet gelogd.

6.1.4 Stap 3: Hardware cursor inbouwen

Wanneer eerst hardware cursor support wordt ingebouwd, kan de volgende stap ermee worden getest.

Zodra de cursorhooks worden geëxporteerd door de driver, zal de app_server deze gebruiken. Dit betekent dat als de hardcursor nog niet werkt, er geen cursor op het scherm wordt getoond. Toch kan met een beetje moeite en enige ‘restarts’ wel worden gewerkt.

Om te beginnen zou het cursor enable/disable register kunnen worden geprogrammeerd. Wanneer ook de eventueel aanwezige voorgrond en achtergrond kleur registers met wit en zwart worden ingevuld, zal waarschijnlijk direct een cursorbitmap ergens op het scherm zichtbaar worden. De bitmap zal ‘random’ zijn, maar wel zichtbaar.

Als nu de positie van de cursor wordt geprogrammeerd, dus de X en Y coordinaat waar deze zich op het scherm bevindt, kan alweer goed worden gewerkt: De random bitmap kan tenslotte nu over het scherm worden bewogen.

Als laatste basisfunctie moet nu de door de app_server aangereikte bitmap op de juiste manier in het geheugen worden geprogrammeerd. Dit kan enig geduld nodig maken, omdat de implementatie van kaart tot kaart nogal kan verschillen:

- Soms wordt normaal videokaart geheugen gebruikt, maar soms ook 'dedicated' geheugen op een aparte plaats benaderbaar via een aantal registers.
- Het referentiepunt van de cursorbitmap kan bijvoorbeeld links-boven of rechts-onder liggen.
- De cursor werkt met vier 'kleuren', in twee bits per pixel. De kleuren zijn (in BeOS) wit, zwart, geinverteerd en doorzichtig. Op een enkele byte geheugen passen dus vier pixels. Van kaart tot kaart verschilt de volgorde van het opslaan van deze pixels. Soms zijn ze achter elkaar geplaatst van links-boven tot rechts-onder. Soms ook volgt als eerste de 'bit 1 plane' data voor alle pixels, en daarna pas de 'bit 0 plane' data (of andersom).

Wanneer normaal videokaart geheugen wordt gebruikt voor de cursorbitmap, kan deze bitmap bijvoorbeeld worden geplaatst op het eerste geheugenadres dat op de kaart bestaat. Omdat in de vorige stap de ruimte voor de desktop daar ook al geplaatst is, zal de app_server het eerste deel van de Desktop overschrijven. Dit is een handig hulpmiddel om te constateren dat de cursordata inderdaad op de verwachte plaats wordt gezet.

Bovendien is dit ook een hulpmiddel voor het testen van de volgende stap: Als die stap werkt, bestaat deze storing niet meer: de Desktop kan dan na de cursorbitmap in het geheugen worden geplaatst.

De cursorbitmap is meestal 64 bij 64 pixels groot in de kaarten. BeOS gebruikt maar 16 bij 16 pixels ervan: de rest dient op 'transparant' te worden ingesteld. De 64 bij 64 pixels à twee bits per stuk nemen in totaal 1024 bytes geheugen in beslag. Als de cursorhardware maar een enkele pixel per byte geheugen ondersteunt (komt ook voor), dan dient 4096 bytes voor de cursordata te worden gereserveerd. Hierna kan dan dus de Desktop ruimte in de kaart worden geplaatst.

Van de cursor zelf moet worden getest of er storingen tijdens verplaatsing optreden. Als dat zo is, moet de programmering van de coördinaat registers gesynchroniseerd verlopen aan de vertical retrace. Meer informatie hierover kan worden gevonden in de beschrijving van de accelerant functie MOVE_CURSOR.
Ook moet worden getest of de cursor stoort bij het veranderen van de bitmap. Hiervoor kan het beste de cursor in BeIDE worden bewogen: de bitmap wordt dan bij elke mouse move overschreven. Bij storingen moet weer een koppeling worden gemaakt met de vertical retrace, zie ook de beschrijving van SET_CURSOR_SHAPE.

Let erop dat de cursor code snel moet zijn: deze code wordt frequent uitgevoerd!

6.1.5 Stap 4: Framebuffer startadres instellen

De laagste paar bits van het startadres worden vaak niet opgeslagen. Dit betekent dat er een minimale stapgrootte (granularity) is voor het zetten van dit startadres. Dit is van belang voor de volgende stap.

Nadat de betreffende CRTC registers geprogrammeerd zijn, zou een eventuele storing ten gevolge van de hardcursor (zie de vorige stap) verdwenen moeten zijn.

Wanneer nu een displaymode op het scherm wordt ingesteld met een hoogte welke groter is dan het zichtbare deel ervan, kan worden getest of het instellen van het startadres goed werkt. Als de cursor naar beneden buiten het scherm wordt bewogen, volgt het scherm de cursor via het zetten van dit startadres.

Controleer of alle bits van het startadres werken door zover mogelijk naar beneden te scrollen in een zo hoog mogelijk virtueel scherm. Dit kan worden ingesteld via BScreen.Setmode(). Zorg ervoor dat de breedte en andere eigenschappen van het scherm ongewijzigd blijven.

Controleer ook of er kortdurende storingen optreden bij het verplaatsen van het scherm. Als dit zo is moet het instellen van het startadres gesynchroniseerd aan de vertical retrace plaatsvinden. Er moet hier wel een timeout worden ingebouwd: Het startadres wordt namelijk ook ingesteld tijdens een SetMode commando. Er zijn op dat moment echter geen vertical retraces!

6.1.6 Stap 5: Framebuffer pitch instellen

De CRTC pitch (offset) registers stellen de afstand tussen twee ‘regels’ van het scherm in. Als dit geprogrammeerd is kan een virtueel scherm worden ingesteld met een grotere breedte dan het zichtbare scherm. De belangrijkste test hier is het zien of de schermuitvoer is zoals verwacht. Als een programmeerfout gemaakt is ‘valt het scherm om’: De tweede regel zit niet recht onder de eerste regel, enzovoorts.

Let erop dat in de MOVE_DISPLAY en MOVE_CURSOR functies de stapgrootte voor het verplaatsen van het scherm juist ingesteld is, afgeleid uit de vorige stap. De stapgrootte is gegeven in pixels, terwijl de vorige stap byte informatie opleverde: de stapgrootte is dus afhankelijk van de ingestelde kleurdiepte.

Er kan nu door het bewegen van de cursor worden getest of het deel buiten het zichtbare scherm ook OK is. Rechts bovenaan bijvoorbeeld staat normaal het BeOS menu. Ook kan nu worden gekeken of de stapgrootte van het bewegen van het scherm is zoals verwacht.

6.1.7 Stap 6: Kleurdiepte instellen

De volgende stap is het instellen van het colordepth register in de videokaart. Dit register bepaalt de kleurdiepte en daaraan gekoppeld het aantal clock cycli dat nodig is om de benodigde data voor een pixel uit het geheugen te halen. Er is bijvoorbeeld voor een 32 bit mode vier maal zoveel data nodig als voor een 8 bit mode.

Nu kan het BeOS Screen Preferences Panel worden gebruikt om te testen of deze functie ook werkt. Wanneer een andere kleurdiepte wordt gekozen dan diegene waarmee is opgestart (via de VBE2 mode dus), dan dienen de pixels op het scherm dezelfde breedte te houden. Als pixels smaller (het scherm ‘herhaalt’ zich dan ook) of breder (ook slechts een deel van het scherm te zien) zijn, dan werkt de functie niet.

De afgebeelde kleuren zullen waarschijnlijk fout zijn: dit is normaal en wordt in de volgende stap verholpen. Het scherm kan bijvoorbeeld foute kleuren bevatten met de juiste helderheid, of alleen donkere, alleen lichtere of alleen zwart/wit achtige kleuren hebben. Eventueel kan worden getest met een andere kleurdiepte in de VBE mode zodat het kleurenpalette anders ingesteld is.

6.1.8 Stap 7: Kleurenpalette instellen

Het instellen van de CRTC palette RAM is nu aan de beurt. In 8 bit kleurdiepte wordt het palette door de app_server ingesteld via SET_INDEXED_COLORS. Bij de beschrijving van deze accelerant functie is een uitgebreide uitleg gevoegd over het uitvoeren hiervan in de driver.

De functie wordt getest met het BeOS Screen Preferences Panel. Wanneer nu een andere kleurdiepte wordt ingesteld, zal het scherm er prima uit blijven zien. Het BeOS Backgrounds Preferences programma is een handig hulpmiddel waarmee de inhoud van de palette RAM kan worden bekeken op grafische wijze. Test alle kleurdiepten om zeker te zijn dat het programmeren van de palette RAM geen fouten bevat.

6.1.9 Stap 8: DPMS inbouwen

Nu dient het in- en uitschakelen van de monitor te worden ingebouwd. Deze functie is belangrijk zodra we de monitor timing gaan (her)programmeren, of de pixelclock gaan aanpassen. Terwijl de timing in de videokaart wordt geprogrammeerd moet de monitor namelijk uitgeschakeld zijn om eventuele beschadiging daarvan te voorkomen. Bovendien worden anders mogelijkerwijze tijdelijke storingen zichtbaar op het scherm en dat staat niet erg netjes.

Het ‘Display Power Management System’ is niets anders dan het in- en uitschakelen van de Hsync en Vsync signalen die naar de monitor worden verstuurd. De monitor gebruikt aan- of afwezigheid van deze signalen als indicatie voor de in te stellen ‘diepte van de slaap’. Beide signalen aanwezig betekent dat beeld moet worden gegeven, terwijl de monitor in de diepste slaapstand wordt gebracht door afwezigheid van beide signalen. Er bestaan ook nog twee tussenliggende slaapstanden: alleen Hsync aanwezig en alleen Vsync aanwezig.

Intern in de driver wordt nog een derde signaal naast de Vsync en Hsync gebruikt. Dit derde signaal schakelt de generatie van de timing in de kaart in en uit. Dit gebeurt met behulp van het 'screen off' bit in de Sequencer 'clocking mode' register op index 1, zoals deze volgens de VGA standaard bestaat. Soms wordt er een combinatie gemaakt met de sequencer 'synchronous reset' conditie (index 0).

Wanneer een kaart het in- en uitschakelen van de synchronisatiesignalen niet ondersteunt, is toch het interne 'derde' signaal nodig om een modeswitch veilig uit te kunnen voeren.

DPMS kan worden getest via de BeOS Screensaver preference applicatie. Stel de activateertijd voor de screensaver zo kort mogelijk in (30 seconden). Het uitschakelen van de monitor kan het beste iets later worden gekozen, bijvoorbeeld na één minuut. Wanneer een monitor wordt gebruikt met volledige DPMS ondersteuning kan gewoon worden afgewacht of de monitor na één minuut uitschakelt om de werking van power management min of meer te controleren. Welke van de drie slaapstanden daadwerkelijk wordt gekozen na één minuut kan meestal niet via de monitor worden gecontroleerd tenzij bijvoorbeeld de kleur van de power-led de slaapdiepte aangeeft.

Het meekijken met de accelerant op een remote computer via een telnet sessie en het tail commando op een logfile kan handig zijn. Zie hoofdstuk 6.2.2 voor meer details hierover.

6.1.10 Stap 9: Refreshrate instellen

Gedetailleerde informatie over de beperkingen van de pixelclock snelheid en dus de refreshates die worden ondersteund, is gegeven in de uitleg van de accelerant functie GET_PIXEL_CLOCK_LIMITS.

Naast de informatie daar gegeven zijn er nog een aantal punten waarop moet worden gelet. Deze punten houden verband met de interne werking van een PLL:

- De VCO heeft een minimaal en een maximaal haalbare frequentie. Het kan zijn dat een VCO in een kaart extra restricties oplegt voor het programmeren van de PLL: niet alle schaal factoren zijn dan mogelijk. Een voorbeeld hiervan is te vinden in de Matrox kaarten en de openBeOS Matrox driver;
- De comparator heeft ook minimale en maximale frequentie restricties voor beide ingangen. Ook dit kan extra restricties opleveren voor het programmeren van de PLL. Een voorbeeld hiervan is te vinden in de Chips en Technologies chipsets, bijvoorbeeld de C&T 69000;
- Op sommige kaarten moet afhankelijk van een gekozen VCO frequentie een feedback filter worden gekozen. Het kan zijn dat zo'n filter bedoeld is voor een gefixeerd frequentiebereik zoals op Matrox G100-G400 kaarten, maar het kan ook zijn dat de fabrikant niet kan garanderen dat een bepaald bereik wordt gehaald. In dit laatste geval moet proefondervindelijk een filter worden gekozen dat op dat ene kaart exemplaar toevallig werkt. De werking van een filter kan worden gecontroleerd aan de hand van de 'lock' informatie van de PLL. Matrox G450 en G550 kaarten werken bijvoorbeeld met dit principe: zie bijvoorbeeld de openBeOS Matrox driver.

Uitleg over de theorie achter de PLL valt buiten het bestek van dit document. Informatie hierover kan worden gevonden in boeken die hoogfrequent electronica voor zendertechniek beschrijven en in diverse grafische chip documentatie, maar dan toegespitst op de desbetreffende chip (sommige functies ontbreken of zijn vast ingesteld in sommige kaarten). Zie de Chips & Technologies CT69000 documentatie voor een voorbeeld.

Het testen van de werking van de PLL kan via controle op het PLL-lock bit in een register als de te besturen kaart deze informatie beschikbaar stelt. Een ander belangrijk hulpmiddel is de computer monitor. Via het OSD kan behalve de instellingen van de monitor ook informatie over de aansturing worden opgevraagd. De refreshrate (de verticale verversfrequentie) en de horizontale verversfrequentie zijn meestal beschikbaar.

De refreshrate welke wordt aangegeven hoeft nog niet precies te kloppen omdat de CRTC timing nog niet wordt geprogrammeerd. De werkelijk actieve CRTC timing (uit het BIOS van de kaart, geactiveerd via de VBE2 mode) kan afwijken van de timing die de driver veronderstelt als zijnde actief. De CRTC timing wordt in de volgende stap onderhanden genomen.

Via de BeOS Screen Preferences Panel kan nu een refreshrate worden gekozen. Controle vindt dan plaats via het OSD van de monitor. Neem de tijd om verschillende instellingen te testen. Storingen bij niet gerespecteerde grenzen worden zichtbaar via trillend beeld, geen beeld, missende pixels in het beeld, erg foute refreshrates en dergelijke.

Opmerking:
Pas op met interne LCD displays van laptops en dergelijke! Deze mogen alleen op 60Hz worden gebruikt. Refreshrate tests dienen plaats te vinden via een externe monitor.

6.1.11 Stap 10: Monitortiming instellen

Het volgende item is het instellen van de CRTC timingsregisters. Deze registers bepalen de grootte van het zichtbare deel van het (virtuele) scherm op de monitor, en de lengte en relatieve plaats van de synchronisatie pulsen en de 'blanking' pulsen. Ook het totaal van alle tijden moet worden ingeprogrammeerd: dit vormt de bruto timing welke samen met de pixelclock de refreshrate bepaalt. (Het zichtbare deel van het scherm zou kunnen worden opgevat als de netto timing.)

De CRTC registers zijn meestal nog op de standaard VGA manier in de kaart gebouwd (met enige uitbreidingen). Soms moeten ze zelfs nog per sé via ISA I/O space worden benaderd. Voorbeelden hiervan zijn de C&T65554 (de latere chips zijn volledig benaderbaar via PCI space) en alle NeoMagic Magicgraph en MagicMedia chips.

Bij het programmeren van dit soort CRTC implementaties moeten waarschijnlijk twee originele standaard VGA 'sloten' worden opengemaakt. CRTC registers op index 0 t/m 7 zijn beveiligd via index $11 (Vsync_end), terwijl de Vsync signalen beveiligd zijn via index 3 (Hblank_end). Merk op dat dit erop lijkt dat er een visieuze cirkel bestaat omdat de registers elkaar lijken te beveiligen. Bij Neomagic kaarten kunnen eerst de registers op index 0-7 worden vrijgegeven, gevolgd door de Vsync registers.

Bijvoorbeeld bij Nvidia TNT en latere kaarten is het mogelijk om de CRTC registers te benaderen via ISA, maar ook via PCI space. Terwijl de register- en bit indelingen identiek zijn via beide toegangswegen, zijn de sloten hier verschillend! Dit voorbeeld is uitgewerkt terug te vinden in de sources van BeTVOut[19].

Nadat de CRTC programmering in de driver opgenomen is, kan worden getest. Hiervoor kunnen diverse resoluties worden ingesteld via het BeOS Screen Preferences Panel. Op de monitor is dan te zien of de zaak werkt. Controleer ook de refreshrate voor diverse resoluties: fouten in de refreshrate wijzen op fouten in CRTC (of PLL) programmering. Een uitgeschakeld scherm kan ook het gevolg zijn van CRTC programmeerfouten. PLL programmeerfouten zullen nooit erg groot zijn, omdat de PLL programmering al is getest bij de vorige stap.

6.1.12 Stap 11: 'Enhanced mode' inschakelen

Dit is de stap waarin we de VBE2 mode gaan verwijderen welke nog 'onder de driver' actief is. Op dit moment wordt de VBE2 mode alleen nog gebruikt om de kaart van een ouderwetse standaard VGA mode (minder dan 256 kleuren met maximaal ongeveer 640x480 resolutie) om te schakelen naar de 'enhanced modes' (modes met 256 kleuren - 8 bit - of meer in minimaal 640x480 resolutie) die worden gebruikt met BeOS.

Enhanced modes hebben over het algemeen als enige de beschikking over de grafische hardware cursor (welke is beschreven in dit document), de acceleratie engine en hardware overlay. De enhanced modes zijn per definitie grafische modes, geen textmodes dus.

De manier van overschakelen hangt van de specifieke kaart af. Bij Matrox bijvoorbeeld noemen ze de enhanced modes 'MGA mode' of 'PowerGraphics' mode.

Om de switch te maken moet zeker gesteld zijn dat de 'enhanced' electronica ingeschakeld is. TVout chips bijvoorbeeld zijn vaak apart van power te voorzien via een configuratieregister in de grafische chip. Ook moeten enige 'standaard VGA' registers worden geprogrammeerd zoals enige sequencer registers in verband met 'clocking modes'.

Het testen van deze stap werkt in gedeelten. Eerst kan worden getest of de gedane omschakelprogrammering de goede werking van de driver met geactiveerde VBE2 mode niet verstoort. Als dat goed gaat kan er nog iets missen, maar (waarschijnlijk) niets verkeerd gedaan zijn tenminste.
Hierna kan de VBE2 mode worden verwijderd en worden gekeken of de driver naar behoren zelfstandig functioneert. Het beeld op de monitor kan zoals meestal ook hier weer nuttige informatie (hints) opleveren.

Opmerking:
Als deze stap succesvol afgesloten is, hebben we de basisdriver compleet! Methoden voor het ontwikkelen van een driver die niet op de VBE2 manier gebaseerd zijn, kunnen hier alsnog in dit stappenplan instappen. De nu volgende stappen kunnen in principe op elke willekeurige volgorde worden gedaan. De hier genoemde volgorde lijkt echter de meest logische.

6.1.13 Stap 12: Acceleratie inbouwen

Als eerste kan het beste de SCREEN_TO_SCREEN_BLIT worden opgezet omdat dit de meest zichtbare versnelling van alle 2D acceleratie functies oplevert. Bij fouten hierin worden storingen zichtbaar bij het verplaatsen van windows en scrollen/pannen in windows (‘wegvallende’, ‘degraderende’ inhoud van de window).

Een goede tweede kandidaat is de FILL_RECTANGLE hook welke wordt gebruikt voor het aanmaken van achtergrondkleuren in windows en op de Desktop. Bij fouten hier worden storingen zichtbaar bij bijvoorbeeld schermwisselingen (achtergrondkleurfouten)

INVERT_RECTANGLE en FILL_SPAN kunnen nu worden opgezet. Deze functies worden net als de eerste twee door de app_server gebruikt, alleen minder vaak.

De laatste twee functies SCREEN_TO_SCREEN_TRANSPARANT_BLIT en SCREEN_TO_SCREEN_SCALED_FILTERED_BLIT worden niet door de app_server gebruikt en zijn dus alleen leuk als extra. Ze kunnen alleen door applicaties via BWindowScreen of via het rechtstreeks laden van de accelerant worden gebruikt. De laatst genoemde functie kan waarschijnlijk niet op 2D only kaarten worden geïmplementeerd omdat deze functie in principe gebruikt maakt van (een deel van) de 3D texture engine.

Het is belangrijk om veel tijd te nemen om zeker te zijn dat alle relevante registers worden geprogrammeerd, anders kunnen problemen onstaan op sommige kaart / moederbord combinaties bijvoorbeeld (‘random’ fouten).
Verder kunnen restricties voor acceleratie afwijken van de CRTC restricties. Bijvoorbeeld de framebuffer pitch. Dit specifieke probleem kan worden opgelost via mode slopspace: meer informatie over dit probleem staat in de beschrijving van de accelerant functie INIT_ACCELERANT.

Het testen van accelerant functies loopt via het werken met BeOS en het goed kijken naar het scherm. De beeldfouten kunnen hints naar het onderliggende probleem geven. Het is handig om de acceleratiefuncties één voor één in te bouwen en één voor één de bijbehorende hooks te exporteren.

6.1.14 Stap 13: Hardware overlay inbouwen

Wanneer de kaart waar de driver voor wordt geschreven de beschikking heeft over een hardware overlay unit, ook wel Backend Scaler (BES) genoemd, kan ondersteuning voor deze hardware in de driver worden ingebouwd. Voor weergave van bewegend beeld video zoals MPEG1, MPEG2 VCD’s en DVD’s is deze functie belangrijk. Niet alleen kan worden volstaan met een veel langzamere CPU wanneer overlay wordt gebruikt, ook de beeld kwaliteit is veel beter. Software output dropt of kopieert meestal pixels wanneer moet worden verschaald, terwijl de overlay unit meestal keurig kan interpoleren. Let erop dat de driver minstens drie overlay bitmaps ondersteunt: dit is namelijk nodig voor dubbel gebufferde videoweergave.

Als eerste kan het beste ongeschaalde output worden geprogrammeerd. Colorkeying kan waarschijnlijk worden uitgeschakeld op de kaart, zodat de ‘bruto’ output zichtbaar wordt op het scherm. De videoweergave wordt nu alleen bepaald door de aan de hardware opgegeven outputwindow coördinaten.

Uitgeschakelde colorkeying is een belangrijk hulpmiddel bij het foutzoeken en het testen van de output op de juiste werking. Wanneer het venster waarin de video wordt afgespeeld bijvoorbeeld van het zichtbare scherm wordt afgesleept met de muis, verlaat in principe ook de inhoud van het venster het scherm. Dit mag echter niet gebeuren bij overlay output: die moet ten dele zichtbaar blijven op het scherm. Voor Matrox kaarten geldt bijvoorbeeld dat altijd een blokje van minimaal twee bij twee pixels moet blijven bestaan. Zodra colorkeying wordt geïmplementeerd, worden deze pixels automatisch onderdrukt. Ook schakelen naar andere workspaces in BeOS gaat dan goed, omdat de colorkey op de andere workspaces niet wordt afgebeeld.

Het is belangrijk goed te testen of de overlay code in orde is. Wanneer het verschalen van de video is ingebouwd is een goede test bijvoorbeeld de volgende:
Schaal het outputwindow op 50% of 150%. Sleep dit window dan langs alle kanten van het scherm, terwijl de helft van het outputwindow buiten het scherm blijft. Zet het beeld hiervoor stil en let op of in het gehele afgebeelde deel van het venster video te zien is, en of de positionering ervan klopt.

Via logging kan worden bekeken of het clippen (afsnijden) van de output goed werkt. Op Matrox kaarten mag geen video buiten het zichtbare gedeelte van het scherm worden ‘getoond’: Dit levert soms ontoelaatbaar hoge belasting voor de kaart op, wat zich onder andere kan uiten in storingen in de Desktop of de videoweergave. Er ontstaan dan bandbreedte problemen waardoor de benodigde data voor de actieve functies niet op tijd uit de videoRAM kan worden gehaald.

Bij afrondingsfouten in het verschalen van de video kunnen bijvoorbeeld één pixel brede storingsstrepen langs de rechterkant of onderkant van de video ontstaan wanneer deze geheel in beeld is: tenminste wanneer links-boven het referentiepunt in de kaart is. Test dit door het venster te verschalen via slepen met de muis.

Wanneer de overlay functonaliteit goed werkt, kan filtering en colorkeying als laatste worden geactiveerd.

Let erop dat de code voor CONFIGURE_OVERLAY snel moet zijn omdat deze bij dubbel gebufferde overlay tot 30 maal per seconde wordt uitgevoerd. Voor overlay hardware welke alleen bereikbaar is via ISA I/O space geldt dus dat het programmeren van de registers in één keer in de kernel driver moet worden gedaan vanwege de trage context switch! Deze situatie geldt bijvoorbeeld voor de NeoMagic MagicGraph en MagicMedia kaarten.

Meer informatie over hardware overlay kan elders in dit document worden gevonden, bijvoorbeeld in hoofstuk 4.1.16: De hardware overlay functies.

6.1.15 Stap 14: Koude start van de kaart inbouwen

Als de specificaties van de kaart gedetailleerd bekend zijn, kan misschien een koude start van de kaart worden opgezet. Dit maakt werken met de kaart als secundaire kaart in een systeem mogelijk . Ook versnelt dit vaak de werking van de kaart omdat chip-core en RAM snelheden in ‘enhanced mode’ vaak hoger zijn dan in de ouderwetse standaard VGA modes waarin de kaart bij primair gebruik tijdens de systeemstart door zijn VGA BIOS wordt geplaatst.

Testen van secundair gebruik van de kaart kan met behulp van een testapplicatie welke de accelerant laadt en test. Als alternatief kan de ‘truc’ met twee videokaarten uit de voorbereidende stap voor het schrijven van drivers worden gebruikt. Deze keer beschouwen we onze driver en kaart als bruikbaar voor secundair gebruik, zodat we een andere, ‘niet ondersteunde’ kaart nodig hebben voor de systeem opstartweergave.

Op deze manier wordt onze driver geladen welke onze kaart gaat (proberen) koud te starten. Terwijl het bootscherm op het primaire display blijft staan, zal de BeOS Desktop op de te testen driver tevoorschijn komen mits de koude start voldoende goed werkt.

Problemen met bijvoorbeeld de hardcursor, acceleratie en overlay kunnen zo wel worden opgelost. Hiervoor is wél goede documentatie nodig van de kaart, of veel geduld.
Display mode problemen (dus helemaal geen beeld) zijn lastiger: een testprogramma kan in dit geval handiger zijn.

6.2 Testen met de driver

Naast het goed kijken naar het door de driver geproduceerde beeld en de geproduceerde timing via het OSD op de monitor, kan het loggen van acties die in de driver worden gedaan het foutzoeken bij het schrijven van de driver sterk vereenvoudigen. Het teruglezen van net geprogrammeerde registers kan handig zijn bijvoorbeeld om te zien of de registers wel beschrijfbaar zijn: denk bijvoorbeeld aan de CRTC locks.

6.2.1 Kerneldriver

In de kerneldriver kan gebruikt worden gemaakt van de in BeOS geïmplementeerde functie dprintf(). De uitvoer van deze functie kan door het systeem naar een serieële poort worden gestuurd als 'debugging' is ingeschakeld.

Het inschakelen van debugging kan op diverse manieren. Tijdens de systeemstart kan debugging bijvoorbeeld via een menu worden aangezet. De output van de kerneldriver wordt dan samen met de output van alle andere systeemonderdelen naar COM1 gestuurd. Met een terminalprogramma kan op een tweede systeem naar deze meldingen worden gekeken.

Een tweede methode voor het inschakelen van debugging is via de 'kernel' configuratiefile in BeOS in de directory /boot/home/config/settings/kernel/. Hierin kan worden opgegeven of debugging aan- of uitgeschakeld moet zijn, en naar welke poort de informatie moet worden gestuurd. Gegadigden zijn COM1 en COM2.

Debugging kan ook in de code van de driver worden in- en uitgeschakeld via de BeOS functie set_dprintf_enabled(bool). Op deze wijze kan ook alleen informatie uit de driver naar buiten worden gestuurd. Een dprintf() commando wordt dan in principe altijd vergezeld van het in- en weer uitschakelen van debugging. Het is wél van belang ervoor zorg te dragen dat de logfunctie uitschakelbaar blijft als via deze methode wordt gewerkt. Tijdens normaal gebruik is loggen ongewenst vanwege bijvoorbeeld de vertraging die het oplevert.

Opmerking:
Het lijkt er helaas op dat deze vorm van debugging bij gebruik van een VBE mode problemen in BeOS oplevert. Het komt voor dat gebruik van dprintf() een hangend systeem oplevert wanneer zo'n mode actief is. De dprintf() uitvoer zelf gaat dan zelfs verloren. Wanneer onverwachte problemen met deze vorm van debugging in de driver ontstaan kan het handig zijn om dit probleem uit te sluiten door debugging uitgeschakeld te houden. Misschien kan dan beter voor een andere methode worden gekozen zoals het zelf aanmaken van een file waarin logmeldingen worden geplaatst. Deze (voor kerneldrivers alternatieve) methode wordt hieronder beschreven.

6.2.2 Accelerant

In de accelerant kan informatie naar een file worden geschreven (in een kerneldriver ook trouwens). Een accelerant kan bijvoorbeeld een eigen logfile aanmaken en die steeds voorzien van een nieuwe regel tekst via een macro als daartoe aanleiding is.

Wanneer de gebruiker nu een terminal sessie opent en een tail commando op de logfile uitvoert, wordt elke nieuwe regel direct op het scherm getoond. Dit kan ook met een remote computer via een telnet sessie worden gedaan. Met behulp van <alt> en de cursor toetsen kan door de uitvoer worden gescrold.

Tail moet hiervoor als volgt worden gestart:
tail –f logfile.txt
logfile.txt is de naam van de logfile. In dit voorbeeld wordt het tail commando gegeven in de directory waar de logfile zich bevindt.

In de accelerant kan beter niet via de kerneldriver worden gelogd! De context switch tussen userspace en kernelspace moet zoveel mogelijk worden vermeden vanwege de traagheid ervan.

Om de accelerant niet te veel te vertragen tijdens normaal gebruik is het handig als de logfunctie uitschakelbaar is. Zeker voor hardware overlay is dit van belang omdat de daarbijbehorende accelerant functie

CONFIGURE_OVERLAY bij dubbel gebufferde overlay tot 30 keer per seconde wordt uitgevoerd. Ook de cursor functies worden zeer frequent uitgevoerd!

6.3 Stabiliteit

Omdat de kerneldriver bij een fout het gehele systeem kan laten crashen en de accelerant bij een fout de app_server kan laten crashen, is het zaak om nauwkeurig te werken. Wat er ook gebeurt, de gebruiker van de computer mag de driver niet laten crashen.

Dit betekent dat de driver nauwgezet moet worden geschreven. Werk stap voor stap: los het ene probleem op voordat aan het volgende wordt begonnen. Voeg functie voor functie aan de driver toe en test uitvoerig. Soms kan het noodzakelijk zijn om programmeeracties onderling enigszins te vertragen omdat de kaart tijd nodig heeft om sommige dingen uit te voeren. Zorg er altijd voor dat er voldoende marge is tussen het vermeende worst case scenario en de geprogrammeerde snelheid: minstens een factor twee. Zelfs programmeer speficicaties verstrekt door kaartfabrikanten bevatten onvolledigheden en fouten!

Variabelen die aan de driver worden aangereikt door een applicatie via de app_server of rechtstreeks dienen te worden gecontroleerd voordat ze worden gebruikt. Het is bijvoorbeeld heel goed mogelijk dat een applicatie probeert hardware overlay op te zetten, maar per ongeluk een NULL pointer of een pointer naar een verkeerde (niet overlay) bitmap meegeeft bij het aanroepen van de CONFIGURE_OVERLAY hook in de accelerant (via BView.SetViewOverlay()).
Als geen controle plaatsvindt op dit soort fouten zal vroeg of laat een applicatie ervoor zorgen dat het gehele systeem plat gaat via de nieuwe driver. Zat de controle wèl in de driver dan zou de gebruiker worden geconfronteerd met een keurige foutmelding betreffende de applicatie en zou de applicatie daarna worden afgesloten.

Het is ook belangrijk ervoor te zorgen dat gedeelten in de driver die niet mogen worden onderbroken door zichzelf of door andere functies in de driver worden afgeschermd via semaforen (critical sections). Een commando verstuurd over een I2C bus bijvoorbeeld mag nooit worden onderbroken voor een ander commando. Het eerste dient te worden afgewacht: deze bus *moet* vrij zijn voordat een nieuw bericht wordt verstuurd.

Wanneer moet worden gewacht op informatie uit de kaart mag een register niet constant worden afgevraagd. Er moet altijd een korte wachttijd worden ingebouwd tussen het herhaald (in een lus) afvragen van een register. Als dit niet wordt gedaan onstaat een zeer hoge piekbelasting van de AGP of PCI bus waardoor de werking van andere kaarten of systeemcomponenten in principe kan worden vertraagd of verstoord. Wachten dient het liefst passief te worden gedaan zodat er geen kostbare CPU tijd verloren gaat: gebruik snooze() in plaats van spin().

6.4 Conclusie

Met behulp van het hier gepresenteerde stappenplan voor het schrijven van een videodriver in combinatie met log functionaliteit kan heel goed een driver worden opgezet. Het stappenplan is geheel getest: zo is het ook tot stand gekomen.

De informatie uit de voorbereidende ‘stap’ stamt uit de BeTVout tijd (Nvidia) toen onder andere de eerste testen met twee videokaarten in de computer gedaan zijn. De openBeOS NeoMagic videodriver is in een periode van twee weken (full-time) opgezet en getest tot en met stap 11 (enhanced mode inschakelen). Dit is gelijktijdig met het schrijven van dit document gebeurd. De latere stappen van het plan zijn eerder al uitvoerig uitgevoerd met de openBeOS Matrox driver.

Bij de genoemde projecten heeft informatie vergaard uit de Linux wereld een belangrijke rol gespeeld. De informatie betrof in alle gevallen achterhaalde programmeerspecificaties voor de gebruikte videokaarten.

7. Conclusie

Het schrijven van videodrivers is leuk om een tijdje te doen. Het is erg leerzaam en (toch) goed te doen wanneer het geheel gestructureerd wordt aangepakt. Soms vergt het schrijven van videodrivers de nodige fantasie van de programmeur, omdat het lastig is sommige (combinaties) van dingen te testen of testbaar te maken.

Ook het achterhalen en begrijpen van de specificaties is soms een uitdaging. Zelfs wanneer wél enige documentatie van een chip- of kaartfabrikant aanwezig is, moet er nog regelmatig worden gepuzzeld. Waarschijnlijk vanwege de grote druk bij de fabrikanten om steeds als eerste met een nieuwe kaart op de markt te komen, zie je dat de domumentatie vaak slecht of onvolledig is.

Samengevat is te stellen: Wie van een uitdaging houdt, moet eens een (BeOS) videodriver gaan schrijven. Met de documentatie en richtlijnen in dit document is al een heel eind te komen. Vooral het opgenomen stappenplan vormt een belangrijk stuk gereedschap. Maar het *blijft* een uitdaging...

Bijlage 1. bronnen voor informatie

Informatie die nodig is voor het programmeren van de kaart is vaak niet (volledig) van de fabrikant te krijgen. Er zijn gelukkig vaak alternatieve bronnen van informatie beschikbaar. Mogelijke bronnen van informatie zijn:
- De fabrikant;
- Linux (of bijvoorbeeld freeBSD[20]);
- Het internet;
- Testen met de kaart voor het verkrijgen van specificaties; en:
- Reverse engineering.

In deze bijlage worden deze informatiebronnen kort besproken.

B 1.1 De fabrikant

Voor veel videokaarten die zijn ingebouwd in moederbord chipsets en videokaarten die geen 3D ondersteunen zijn de specificaties vaak vrij beschikbaar van de producent.
Fabrikanten zoals Matrox en ATI willen ook nog wel eens gedeeltelijk meewerken aan het verstrekken van informatie. De informatie mag dan niet worden doorgespeeld aan derden: er is een registratie van de programmeur nodig waarmee hij of zij aangeeft dit niet te zullen doen: Dit is de 'non-disclosure agreement'.
Er zijn helaas ook fabrikanten die helemaal niet mee willen werken aan het verstrekken van informatie aan de 'kleine' programmeur. Het meest bekende voorbeeld hiervan is Nvidia.

Documentatie is vaak onvolledig of niet te krijgen omdat de fabrikanten bang zijn voor de concurrentie en bij sommige delen van de kaart (TV in- en uitgangen en codecs) waarschijnlijk voor de filmindustrie. Tijdsdruk in de videokaart industrie speelt ongetwijfeld ook een rol: documentatie voor ontwikkelaars wordt vaak samengesteld uit documentatie die intern beschikbaar is, maar waarvan sommige delen niet naar de betreffende groep ontwikkelaars mag worden doorgegeven.

Keiharde fouten in de wèl beschikbare kaart documentatie komen helaas ook geregeld voor. Wanneer documentatie van verschillende generaties op elkaar lijkende chips beschikbaar is, kunnen soms gelukkig goed elkaar tegensprekende verhalen over hetzelfde deel van de chips boven water worden gehaald.

In de gevallen waarin niet alle informatie beschikbaar is over een kaart moet vaak worden gegrepen naar een alternatieve tweede bron van informatie. Deze situatie komt vaak voor.

B 1.2 Linux

Het probleem van missende informatie speelt al veel langer bij alternatieve opensource operating systems zoals Linux. Met dit systeem zijn ook veel meer mensen bezig dan met bijvoorbeeld BeOS.

Voor BeOS programmeurs is dit een prettige situatie, omdat veel informatie uit de Linux sources kan worden verkregen. De grafische drivers die voor BeOS nodig zijn, zijn te vinden in de XFree86 sources[21]. Text-mode drivers en alternatieve grafische drivers zijn soms te vinden in de Linux kernel sources[22].

De XFree86 driver sources zijn een mooie bron van informatie. De grafische drivers zijn redelijk gestructureerd opgebouwd in verschillende files. Er zijn bijvoorbeeld vaak aparte files voor standaard driver functies (initialisatie en mode setup), acceleratie en hardware overlay. Wanneer een kaart letterlijk voortbouwt op standaard VGA, wordt een vga_hardware submodule door de Linux driver ingeladen en gebruikt om de functies hierin uit te voeren. Als dit het geval is bewijst een boek over oude 'standaard' VGA kaarten grote diensten bij het schrijven van de BeOS driver.

Iets dat op een zelfde manier als standaard VGA wordt ondersteund in Linux is gebruik van de DDC channels. Hiervoor wordt ook een submodule ingeladen die het werk doet. Eigenlijk kan BeOS hier wel iets van leren: Een standaard VGA en DDC support library zou de programmeur van drivers werk kunnen besparen.

Als geen gebruik van DDC in de driver voor BeOS wordt ingebouwd (waarschijnlijk niet), dan kan dit deel in de Linux driver uiteraard buiten beschouwing blijven.

Wanneer een Linux driver bestaat voor de videokaart die gaat worden voorzien van een BeOS driver, kan deze Linux driver worden gebruikt voor het vaststellen van de programmeer specificaties van de kaart. Hiermee kan dan een 'standaard' BeOS driver worden geschreven die bovendien niet per sé als GNU software hoeft te worden uitgebracht.

Het letterlijk 'porten' van een Linux driver wordt niet geadviseerd: de verschillen tussen beide systemen zijn toch wel groot.

Wanneer van een Linux driver bekend is wie de auteur is, is het ook zeker de moeite waard om eens contact met hem of haar op te nemen. De auteurs zijn gelukkig vaak geneigd om welwillend informatie te verstrekken over hun drivers, zelfs als de informatie ook in de driver zelf te vinden is. Extra uitleg over het hoe en waarom van de code kan een goede hulp zijn bij het juist interpreteren ervan.

Het leuke van contact met een andere auteur is bovendien dat op een bepaald moment waarschijnlijk ook nieuwe informatie *aan* die persoon kan worden gegeven. Twee personen ontdekken vaak meer dan één.

B 1.3 Internet

Een zoekopdracht voor informatie over een kaart in een algemene zoekmachine kan nooit kwaad. Het zou niet de eerste keer zijn dat benodigde informatie uit een andere hoek komt dan werd verwacht. Het zou tenslotte kunnen dat ergens op de wereld iemand met een specifieke, nog onbekende driver bezig is.

Naast de normale videodrivers worden soms ook drivers voor delen van een kaart geschreven, of drivers toegespitst op een bepaald gebruik. Zo bestaan bijvoorbeeld specifieke TVout oplossingen voor Nvidia kaarten. BeTVout (voor BeOS) en TVtool (voor Windows) zijn hier goede voorbeelden van. Deze drivers werken parallel aan de normale videodriver op een kaart en hebben deels dezelfde functionaliteit.

Ook drivers die zich nog in een vroeg stadium bevinden kunnen soms ook via een zoekmachine worden opgespoord. Dit soort Linux drivers bijvoorbeeld zijn meestal nog niet in de source-tree opgenomen.

Natuurlijk is het ook mogelijk dat officiële informatie toch beschikbaar is, maar niet bij de verwachte bron. Een fabrikant kan failliet gaan, of worden overgenomen bijvoorbeeld. Chip specificaties van Chips & Technologies kaarten moeten bijvoorbeeld worden gezocht bij Asiliant Technologies.

B 1.4 Testen voor specificaties

Wanneer geen kaart specifcaties beschikbaar zijn en wordt gewerkt met een Linux driver als bron voor de specificaties, kan het beste zelf worden geverifieerd in hoeverre de veronderstelde specificaties in de Linux driver juist zijn.

Drivers geschreven voor Linux hebben namelijk vaak de neiging om ervan uit te gaan dat de gebruiker technisch onderlegd is en weet wat hij of zij doet. Range-checking wordt bijvoorbeeld in Linux niet altijd toegepast. Ook het begrenzen van driver instellingen op de maximale prestaties van een kaart wordt soms niet uitgevoerd.

In Linux is deze instelling waarschijnlijk prima vanwege de meer technische doelgroep die het heeft, maar in BeOS hoort deze instelling eigenlijk niet thuis. BeOS behoort te kunnen worden gebruikt door iedereen, zonder dat specifieke technische kennis nodig is. Het OS behoort de gebruiker daarom zoveel mogelijk te behoeden voor fouten.

voorbeeld: CRTC startadres
Wanneer bijvoorbeeld een register op zijn bitbreedte moet worden gecontroleerd zodat juiste range-checking in de driver kan worden toegepast, kan het beste bit voor bit worden getest. Het CRTC startadres bijvoorbeeld kan heel eenvoudig worden gecontroleerd via een zo smal mogelijk, maar zo hoog mogelijk virtueel scherm. Wanneer nu de laagste colorspace wordt ingesteld, kan via het naar beneden bewegen van de cursor worden gecontroleerd in hoeverre de bits van het startadres daadwerkelijk door de kaart worden overgenomen. Met de genoemde instellingen wordt het CRTC startadres onderin het scherm zo hoog mogelijk ingesteld en worden dus de meeste bits ervan gebruikt.

Als onverhoopt tijdens het naar beneden scrollen van het beeld de inhoud van het beeld terugspringt naar het bovenste deel van de virtuele ruimte, wordt het aantal beschikbare bits in het CRTC startadres register overschreden.

Als dit wordt geconstateerd (en het wordt niet veroorzaakt door een programmeerfout), kan het opzetten van een virtueel scherm bijvoorbeeld worden begrensd op de range van deze ‘variabele’ via de accelerant functie PROPOSE_DISPLAY_MODE.

Het klinkt misschien onlogisch dat niet het gehele kaart geheugen voor het afbeelden van een beeldscherm kan worden gebruikt, maar toch komen dit soort begrenzingen in de praktijk voor!

voorbeeld: de PLL
Een ander voorbeeld van een in principe goed testbaar onderdeel van de videokaart is de VCO gebruikt in de diverse PLLs waarmee de RAM, core en DAC (pixelclock) snelheden worden ingesteld. Tenminste, als deze PLLs een ‘lock’ indicatiebit beschikbaar stellen aan de driver.

Als dit zo is, kan via een testversie van de driver een frequentie sweep worden gemaakt om vast te stellen wat de onder- en bovengrens van het bereik van een VCO is. Als uit deze test blijkt dat niet alle theoretisch mogelijke frequentie instellingen daadwerkelijk worden gehaald kan de verkregen informatie worden gebruikt voor het begrenzen van dit onderdeel in de driver.

Omdat een fabrikant ‘per definitie’ niet in staat is om alle kaarten exact dezelfde specificaties te geven, moet een marge worden ingebouwd bij deze begrenzing. Het beste kunnen hiervoor minstens drie kaarten worden getest. Als nu het worst case scenario wordt gebruikt, en daar wordt nog eens een marge van 5-10% extra op ingebouwd, dan zit het meestal wel goed.

Met deze begrenzing kan bijvoorbeeld worden voorkomen dat sommige kaart exemplaren bij sommige instellingen van de refreshrate allerlei beeldstoringen geven.

Het uitvoeren van VCO sweeps heeft de Matrox driver bijvoorbeeld zichtbaar in ‘kwaliteit’ laten toenemen: De specs voor dit onderdeel waren namelijk niet vrijgegeven door de fabrikant voor alle door de driver ondersteunde kaarten.

B 1.5 Reverse engineering

Reverse engineering is het achterhalen van de broncode van een programma wanneer alleen de uitvoerbare versie van de code beschikbaar is. De achterhaalde broncode kan vervolgens worden gebruikt om specificaties van de te besturen kaart te achterhalen.

Reverse engineering wordt vaak als verboden betiteld. Wanneer het doel van reverse engineering echter het verkrijgen van compatibiliteit van een videokaart met een nieuw OS betreft bijvoorbeeld, is dit klaarblijkelijk niet het geval. Dit blijkt bijvoorbeeld uit sectie 1201(f) getiteld ‘Reverse Engineering’ van het Amerikaanse DMCA (Digital Millennium Copyright Act) uit de United States Code van het U.S. house of Representatives [23]. Het blijft echter wel

oppassen geblazen: de interpretatie van de regels verschilt nogal eens. Ook zijn van land tot land verschillen in wetgeving van toepassing.

De meest belangrijke kandidaten voor reverse engineering zijn het videokaart BIOS en Windows drivers.

videokaart BIOS
Het videokaart BIOS is vaak relatief compact. Alleen de basisfuncties voor een videodriver kunnen over het algemeen worden achterhaald, omdat alleen deze functies in het BIOS zijn ingebouwd.

Het betreft hier in principe de stappen 4 (framebuffer startadres instellen) tot en met 11 (enhanced mode inschakelen) uit het stappenplan in dit document. Soms kan ook basisfunctionaliteit voor TVout in het BIOS worden gevonden.

Ook het koudstarten van de videokaart is altijd ingebouwd in het BIOS: dit is stap 14 uit het stappenplan. Het koudstarten is wel erg kaarttype specifiek. Een identieke kaart met bijvoorbeeld alleen langzamer geheugen heeft al een aangepaste startprocedure: de RAM snelheid moet tenslotte anders worden ingesteld.

Wanneer een koude start in een driver opgezet moet worden aan de hand van informatie uit een kaartBIOS kan het beste worden gezocht naar een pointer op een vaste plaats in het BIOS welke verwijst naar een ‘struct’ in dit BIOS waar uiteindelijk de specificaties voor het koudstarten van de kaart vermeld staan. Op deze wijze kan een universele koude start voor de ondersteunde kaarten worden ingebouwd. Bij Matrox kaarten is dit systeem van registratie van specificaties bijvoorbeeld in het BIOS ingebouwd: de driver maakt er dankbaar gebruik van.

De meer geavanceerde functies voor de hardware cursor, acceleratie en hardware overlay tenslotte zijn uit het BIOS niet te achterhalen omdat ze daar niet ingebouwd zijn.

Opmerking:
De VBE functies tot en met versie 2 zijn geschreven voor realmode executie. Dit betekent dat deze functies niet vanuit een protected mode driver (zoals BeOS drivers) kunnen worden aangeroepen. Het BIOS kan in de BeOS driver tijdens run-time dus alleen worden gebruikt voor het achterhalen van specificaties voor het koud starten van de kaart. De eigenlijke koude start dient dus door de driver *zelf* te worden uitgevoerd.

VBE2 kent al wel een protected mode hook, maar deze is slecht gedefinieerd en wordt daardoor zelden toegepast.

VBE versie 3 (en later) ondersteunt *wel* goede protected mode executie. In principe kan een BeOS driver dus *wel* gebruik maken van een VBE3 hook in het BIOS indien aanwezig. Het koudstarten van de videokaart wordt door VBE3 echter nog *niet* ondersteund. Alleen display_modes kunnen vanaf VBE3 op een vrij flexibele manier tijdens run-time door het BIOS worden ingesteld.

Windows drivers

In een Windows driver is in principe elk aspect benodigd voor een BeOS videodriver terug te vinden, en meer. Naast informatie over de basisfuncties die ook beschikbaar zijn in het kaartBIOS valt hier dus informatie over de meer geavanceerde functies te vinden zoals voor de hardware cursor, acceleratie en hardware overlay.

Informatie over het *proces* van reverse engineering valt buiten het bestek van dit document.

Bijlage 2. begrippenlijst

AGP
Accelerated Graphics Port. AGP is een soort extensie op de oudere PCI bus. AGP kaarten zijn ook op de PCI wijze in software te gebruiken. AGP is geintroduceerd om de problemen bij PCI voor grafische kaarten weg te nemen. AGP heeft een databus breedte van 32bit zoals de standaard PCI bus, maar de clocksnelheden liggen belangrijk hoger: standaard werkt AGP op 66Mhz terwijl PCI op 33Mhz werkt, en kent AGP naast de 1x ook een 2x, 4x en tegenwoordig een 8x modus.
Terwijl PCI een theoretisch maximum transferrate heeft van 133Mbyte/sec, kent AGP in 1x mode reeds 266Mbyte/sec, oplopend tot ruim 2Gbyte/sec in 8x modus.

Verder heeft AGP heeft een sterkere voedingsaansluiting dan PCI sloten en hebben AGP kaarten directe toegang tot het systeemgeheugen. Bovendien wordt de bandbreedte van een PCI bus gedeeld over alle insteekkaarten die op de controller zijn aangesloten (meestal max. 6 kaarten), terwijl de AGP poort een eigen ‘dedicated’ controller heeft.

De AGP poort is aangesloten op de North-bridge controller, terwijl PCI is aangeloten op de South-bridge controller van de moederbord chipset. De North-bridge controller is direkt op de CPU en systeemgeheugen aangesloten, terwijl de South-bridge controller aangesloten is via een chipset-specifieke, relatief trage bus op de North-bridge controller. Deze bus is ook een potentiële kandidaat voor een bottleneck.

API
Application Programming Interface. De programmeer interface die applicaties tot hun beschikking hebben om contact te maken met het operating systeem en de daaronder liggende drivers.

backend scaler
Zie hardware overlay.

colorkey
De colorkey is de sleutel die bepaald of de videokaart de output van de backend scaler toont, of dat de output van de videoram buffer waarin de Desktop getekend is wordt getoond. Deze sleutelwaarde is een van te voren gedefinieerde kleur die op de Desktop kan worden getekend om de omschakeling te maken. Colorkeying wordt meestal gebruikt voor hardware overlay.

CRTC
Cathode Ray Tube Controller. Dit is een groep registers in een logische unit binnenin een VGA kaart die de besturing van de beeldbuis voor zijn rekening neemt. De plaats van de electronenstraal en de snelheid van het scannen van het beeld wordt hierdoor bestuurd. De synchronisatie signalen zijn de kanalen waarlangs deze besturing plaatsvindt.

DAC
Digital to Analog Converter. Dit is een functie in een grafische kaart die de digitale informatie uit het video geheugen omzet in een analoog signaal. In feite zijn er drie DACs in één: één voor elke basiskleur. De drie signalen worden naar de drie electronenstralen gestuurd in een beeldbuis kleurenmonitor om de intensiteit ervan te bepalen. Hierdoor worden uiteindelijk de patronen gegenereerd die we als ’het beeld’ omschrijven.

DDC
Display Data Channel. Dit is een serieële poort in de VGA connector waarmee de eigenschappen van de aangesloten monitor kunnen worden opgevraagd. Informatie zoals merk en type van de monitor, de maximale en de voorkeurs resolutie, de minimale en de maximare refreshrates en dergelijke kan in principe worden opgevraagd.

directX
Microsoft’s tegenhanger voor de openGL standaard. Deze standaard is niet open, zodat hij alleen op Windows systemen wordt gebruikt. Omdat Microsoft een machtige’partij’ is, zorgen de meeste grafische kaart producenten

ervoor dat hun kaarten directX ondersteunen. Hierdoor raakt openGL langzamerhand wat op de achtergrond: ook de mogelijkheden met directX zijn tegenwoordig groter.

DirectX ondersteunt naast de vermelde 3D acceleratie ook andere features gebruikt voor ‘gaming’ en dergelijke zoals audio.

DMA
Direct memory access. Hiermee is het mogelijk voor diverse hardware om data van of naar het systeemgeheugen (of een ander stuk hardware) te transporteren zonder tussenkomst van de CPU. De CPU kan zich dan met andere taken bezighouden zodat het totale systeem sneller is.
Elk moederbord heeft twee DMA controllers aan boord die voor deze kopieertaak kunnen worden ingezet. Terwijl ISA kaarten daar soms gebruik van maken, is het voor PCI en AGP kaarten niet langer nodig. De PCI (AGP) controller combineert zelfstandig toegangsaanvragen tot aaneengesloten reeksen adressen zodat een soort vervangend DMA systeem ontstaat. Hiervoor is geen programmering nodig: het is in de hardware van de controller ‘ingebakken’ en gedraagt zich transparant.

double buffering
Terwijl één buffer (gevuld met een plaatje) op het scherm wordt getoond, wordt een andere buffer op de achtergrond gevuld met een nieuw plaatje. Wanneer het nieuwe plaatje af is worden de buffers afgewisseld en begint het proces opnieuw.

Op deze manier kan rustiger worden getekend dan wanneer slechts een enkele buffer beschikbaar is omdat het opnieuw vullen met een plaatje bij gebruik van een enkele buffer tijdens de korte vertical retrace moet plaatsvinden om storingsvrije weergave te garanderen.

Met double buffering is dus veel meer tijd voor het vullen beschikbaar, namelijk de tijd die het kost één keer het beeld te laten zien plus de vertical retrace tijd.

Bij double buffering worden meestal niet twee, maar drie buffers gebruikt zodat de tekenende applicatie geen tijd (of tenminste minder tijd) kwijt is aan het wachten totdat de afgebeelde buffer vrijkomt zodat met overschrijving met een nieuw plaatje kan worden begonnen.

Er is in feite dus een circulaire buffer met drie bitmaps.

DPMS
Display power management system. Modernere VGA monitoren ondersteunen power management. Power management biedt twee belangrijke voordelen: energie besparing en verlenging van de levensduur van de monitor. Screensavers zijn tegenwoordig dus in feite ‘screendestroyers’, tenzij ze de monitor uitschakelen.

DPMS werkt via de beide synchronisatielijnen naar de monitor: Vsync en Hsync. Wanneer één of beide signalen ‘wegvallen’, is power save ingeschakeld. Er zijn 4 stadia mogelijk: van ‘aan’ tot ‘bijna uit’.

driverinterface
Het totaal van de software interface van het OS naar de driver.

filteren
Zie interpoleren.

game kit
Be verdeelde de BeOS API in kits om het geheel overzichtelijk te houden. De game kit is daar één van. Deze kit is bedoeld als hukp voor het programmeren van spelen en dergelijke.

geheugen refresh
Omdat het meest gebruikte soort RAM geheugen vanwege de lage prijs dynamisch geheugen is, moet de inhoud van het geheugen regelmatig worden ververst. Als dit niet gebeurt gaat de inhoud van het geheugen verloren omdat dynamisch geheugen zijn inhoud maar korte tijd (milliSeconden) vast kan houden (capacitieve werking).

Geheugen refresh maakt gebruik van de adresbus zoals normale toegang naar het geheugen dit ook doet: Het uitvoeren van de refresh kost enkele procenten van de beschikbare bruto bandbreedte.

granulariteit
Stapgrootte. De granulariteit van een coördinaat is het grootste getal waardoor de coördinaat waarden deelbaar zijn zodat de uitkomst een geheel getal blijft.

In videokaarten wordt de coördinaat gedeeld door de granulariteit als coördinaat-waarde in een register opgeslagen, waardoor deze granulariteit dus ontstaat.

hardware overlay
Een 'truc' methode om video goed te kunnen afspelen in computers zonder dat daarvoor enorm veel rekenkracht van de systeem processor nodig is. In het algemeen is het zo dat dingen die veel processorkracht kosten als ze in software zouden worden uitgevoerd, vaak in een stuk speciale hardware worden ondergebracht.
Naarmate de processoren sneller worden, wordt de behoefte aan 'trucs' kleiner. In de praktijk wordt dit vaak bevorderd doordat minder hardware minder geld kost: Een bekend voorbeeld hiervan zijn de soft- of winmodems.

hook
Een hook is een 'standaard entry point' voor software. Een aantal standaard routines wordt in de driver geimplementeerd. De ingang naar zo'n standaard routine is de hook.

interface kit
Be verdeelde de BeOS API in kits om het geheel overzichtelijk te houden. De interface kit is daar één van. Deze kit wordt gebruikt voor algemene besturing van het OS.

interlaced mode
Het beeld wordt in twee 'halve' beelden verdeeld alvorens op het scherm te worden getoond: een 'even' en een 'oneven' beeld ('fields' genoemd). Het even beeld bestaat uit de 'even' horizontale scanlijnen, terwijl het oneven beeld bestaat uit de tussenliggende 'oneven' scanlijnen. Het oorspronkelijke beeld wordt nu in twee stappen getoond: Eerst wordt het ene halve beeld getoond, dan het andere halve beeld.

Omdat beide halve beelden nu *na* elkaar worden vertoond, kunnen de beelden in de tijd ook *na* elkaar zijn 'opgenomen'. Meestal is dit ook het geval.

Interlaced mode wordt gebruikt voor het televisie systeem om op bandbreedte te besparen. Zo worden slechts het halve aantal beelden getoond, terwijl het menselijk oog dit ziet als het volle aantal beelden waardoor flikkering tot een minimum wordt beperkt.

Oudere grafische kaarten en monitoren voor de PC kenden dit systeem ook voor weergave van het computerscherm: Dit was goedkoper dan het betere progressive scan systeem.

interpoleren
Ook wel 'filteren' genoemd.

Wanneer een beeldpunt wordt geïnterpoleerd omdat er geen daadwerkelijke informatie voor aanwezig is, wordt het (gewogen) gemiddelde bepaald van het vorige beeldpunt en van het volgende beeldpunt om een zo 'glad' mogelijke weergave te bevorderen.

Wanneer interpolatie bij verschalen van video op een computer wordt gebruikt, voorkomt dit moiré patroon vorming op het beeld.

ISA
Industrial standard architecture. De ISA bus is de extensiebus zoals deze algemeen geaccepteerd was voordat PCI geintroduceerd werd. Oudere PCI systemen hebben ook nog steeds enkele ISA sloten tot hun beschikking.

De oorspronkelijke ISA standaard werd gebruikt in PC/XT computers, en had een 8 bit databus, 64Kbyte I/O adresruimte en werkte op 8.33Mhz. De bus werd met de introductie van de 286 uitgebreid naar een 16bit databus.

Veel PCI en zelfs AGP kaarten zijn nog steeds benaderbaar te maken via de ISA (I/O) space. Configuratie hiervan verloopt via PCI configuration space.

koude start van de videokaart
De videokaart wordt van de 'grond' af aan geïnitialiseerd. De geheugentype en grootte wordt bepaald, de geheugen snelheid wordt ingesteld, memory refresh wordt geactiveerd, en de chip 'core' snelheid wordt ingesteld. Tevens worden hiervoor de nodige hardware componenten op de kaart geactiveerd. Dit zijn alle zaken die in principe eenmalig, dus alleen tijdens de koude start worden uitgevoerd.

little endian
Als een datawoord meer data bevat dan in één keer over de databus kan (of in één geheugenlokatie past), moet het woord in delen worden geknipt en na elkaar worden verstuurd (of geplaatst). Little Endian betekent dat eerst het minst belangrijke deel (LSB) wordt verstuurd en daarna het meest belangrijke deel (MSB). Big Endian doet het net andersom.

mappen van resources
Naast I/O toegang (welke via speciale commando's werkt) naar registers in een kaart bijvoorbeeld, is het ook mogelijk om de registers benaderbaar te maken alsof ze 'normaal' systeemgeheugen zijn. Op deze wijze kunnen de registers veel sneller, zonder het gebruik van speciale I/O commando's (die meestal alleen in de kernel van een OS beschikbaar zijn) benaderen.

openGL
Sun's open standaard voor hardware 3D acceleratie. Omdat deze standaard open is, wordt hij op veel platformen ondersteund.

OSD
On Screen Display. Moderne VGA monitoren zijn in te stellen via een OSD. Dit OSD kan tevens resultaten van metingen verricht aan de timingsignalen komende vanaf de ingang door de monitor laten zien. Dit zijn de refreshrate (verticale snelheid), horizontale snelheid (in Khz) , en soms de pixelclock snelheid.

PCI
Pheripheral component interconnect. Zie AGP.

PCI configuration space
PCI (AGP) kaarten hebben alle een aantal configuratie registers welke zich in de configuration space bevinden. Deze configuratie registers bepalen waar de eigenlijke resources van de kaart (zoals registers en video geheugen) op de bus zullen verschijnen wanneer ze worden geactiveerd.
Bij een systeemstart zijn alle kaarten gedeactiveerd: activatie vindt plaats nadat configuratie heeft plaatsgevonden. Activatie vind ook plaats via de configation space. Als een kaart gedeactiveerd is, is alleen de configuration space bereikbaar op een gestandariseerde manier.
Voor nieuwere ISA kaarten bestaat een vergelijkbaar systeem: de PnP standaard. Oudere ISA kaarten zijn geconfigureerd via jumpers en zijn altijd geactiveerd.

POST
Power On self test. Deze test wordt door elk moederbord verricht na het inschakelen van het systeem. De eigen componenten worden getest om te zien of het systeem in orde is. Naast de ouderwetse POST meldingen naar de ISA I/O space poort $80 worden tegenwoordig ook meldingen op het scherm gezet. Soms is een audio POST reporter ingebouwd. Een stem laat dan de resultaten van de POST horen.
POST is een handig hulpmiddel om fouten in het moederbord op te sporen, hoewel meestal het geheel vervangen van het moederbord noodzakelijk geworden is door de hoge integratie van de componenten.

progressive scan

Bij een (moderne) computermonitor wordt deze manier van scannen door de electronenstraal toegepast. Het volledige beeld (‘frame’ genoemd) wordt in één keer van links-boven met horizontale lijnen naar rechts-onder afgebeeld in licht op de beeldbuis. Interlaced mode is het ‘tegenovergestelde’ van progressive scan.

refreshrate
De refreshrate gebruikt in een display mode geeft aan hoe vaak per seconde het beeld op de monitor wordt ververst. Hoge refreshrates zijn populair omdat het resulterende beeld minder of in het geheel niet flikkert voor het menselijk oog.
Het menselijk oog kan tot ongeveer 30 bewegende beelden per seconde zien flikkeren: Stilstaande beelden zoals op een computer gebruikt kan men zien flikkeren tot ongeveer 70 beelden per seconde: 70Hz. Refreshrates boven de 70 Hertz worden dan ook als ergonomisch beschouwd.

Deze uitspraak geldt niet voor moderde (TFT) LCD schermen: die kunnen met 60Hz prima worden aangestuurd, omdat ze intern het beeld bufferen en continue aan de kijker laten zien. Er is dus een statisch beeld dat perfect stabiel is.

single buffering
Zie ook double buffering. Single buffering wordt alleen toegepast indien niet voldoende videokaart geheugen beschikbaar is om meer dan één bitmap voor hardware overlay aan te maken zoals vaak het geval is op laptops. Omdat ook met single buffering gebruik kan worden gemaakt van de hardware overlay unit of backend scaler, wordt toch een behoorlijke winst behaald in videokwaliteit en verminderde CPU belasting.

slopspace
Als een coördinaat-waarde niet voldoet aan de granulariteit die eraan wordt gesteld, dan wordt de coördinaat-waarde naar boven afgerond zodat wel aan de granulariteit wordt voldaan. Het verschil tussen de gevraagde en de verkregen coördinaat-waarde wordt apart bewaard ter referentie voor de daadwerkelijk gebruikte ‘ruimte’.

TSR
‘Terminate and Stay Ready’ program. Dit soort programma’s betreft vaak een soort drivers voor DOS. Soms was in het DOS tijdperk bijvoorbeeld de grootte van de BIOS-ROM in videokaarten niet groot genoeg om alle functies te bevatten. Of kwamen later updates uit die functionaliteit toevoegden aan de kaart. Een TSR kon dan uitkomst brengen: voor de gebruiker leek het erop alsof het BIOS uitgebreid was zodra deze file geladen was. Op deze wijze kon bijvoorbeeld VBE 2 support aan VBE 1.2 compatible kaarten worden toegevoegd.

unified driver
Een driver die een hele serie, meestal na elkaar door de fabrikant op de markt gebrachte videokaarten ondersteunt.

VBE
VESA BIOS Extensions. Naast de standaard in het VGA BIOS aanwezige grafische- en tekstmodes, werden later grafische modes geintroduceerd voor modes die niet in de VGA standaard voorzien waren. Om nu applicatieprogrammeurs op eenvoudige wijze toegang tot deze extra modes te geven, werd een standaard door de VESA organisatie bedacht die dit mogelijk maakte zonder gebruik van specifiek voor een kaart geschreven drivers.

Deze nieuwe standaard maakt gebruik van de zogenaamde VGA BIOS INT10 (hex) hook en conventies om de modes te activeren. Omdat de grafische kaarten steeds hogere resoluties en kleurdiepten aankonden, moest ook VBE regelmatig worden uitgebreid. Zo ontstonden diverse versies van het VBE: 1.0, 1.2, 2.0 en tegenwoordig 3.0.

Versie 3.0 is de eerste versie van VBE welke op een bruikbare manier uitvoering van de VBE modes in protected mode toestaat. Hiervoor is een speciale ‘protected mode hook’ geintroduceerd, welke (een aantal van) dezelfde modes kan instellen als mogelijk is via de oude ‘realmode’ hook: INT10.

VCO
Voltage controller oscillator. Dit is in de context van dit document een onderdeel van een PLL. Het betreft een ‘frequentiebron’ waarvan de frequentie middels een aantal registers (de ‘delers’ of ‘scalers’) kan worden ingesteld.

vertical retrace

Bij de opbouw van een beeld op een beeldbuis (CRT) monitor wordt een electronenstraal over de oppervlakte van de buis bewogen met hoge snelheid, waardoor het beeld ontstaat. De beeldbuis wordt ‘gescand’ van links naar rechts en van boven naar onder met bijna horizontale lijnen. Als de straal rechtsonder in het beeld aankomt, wordt hij uitgeschakeld en weer snel naar de linker-bovenkant van het scherm gestuurd.

Deze terugstuur actie is de vertical retrace. Tijdens de vertical retrace worden geen beeldpunten opgewekt (omdat de straal uit staat) zodat het beeld kan worden verwisseld of gestoord zonder dat dit te zien is. Het beeld is stabiel voor het menselijk oog omdat het beeld steeds snel na elkaar wordt gescand, en de beeldbuis bovendien een nagloeitijd heeft.

warme start van de videokaart
De driver gaat ervan uit dat de kaart al een koude start heeft gemaakt. Alleen de benodigde display mode wordt ingesteld, waaronder de pixelclock van de DAC. De typische koude start initialisatie wordt niet gedaan. Eventuele benodigde informatie verzameld door het VGA BIOS tijdens de koude start zoals de grootte van het geheugen, wordt uit kaartregisters gelezen welke tijdens deze koude start zijn geinitialiseerd. Soms kan de benodigde informatie ook (rechtstreeks) uit het VGA BIOS worden gelezen.

In het kaartBIOS kunnen de bij de productie bepaalde gegevens zijn voorgedefinieerd.

YCbCr411
Zie YCbCr422. In de colorspace YCbCr411 wordt van elke pixel de helderheidsinformatie opgeslagen (4 bekeken pixels), terwijl van de kleurinformatie één op de vier pixels wordt opgeslagen (1 uit de 4 bekeken pixels).

YCbCr422
Een colorspace met compressie welke wordt gebruikt voor de opslag van video. Oorspronkelijk werd de gebruikte compressietechniek ingevoerd voor het kleurentelevisie signal CVBS (Composite Video Baseband System) wegens beperkte bandbreedte van de bestaande zenders en TV toestellen. De kleurinformatie moest op compatible wijze worden verzonden.

In een TV toestel worden de beelden opgebouwd uit drie basiskleuren. rood (R), groen (G) en blauw (B). Rood, groen en blauw gemengd licht levert namelijk de ‘kleur’ wit op mits in de juiste verhouding gemengd. Wanneer deze basiskleuren in de verhouding gemengd zijn dat ze wit op leveren kunnen alle grijstinten tussen zwart en wit worden opgewekt door variatie van de gezamelijke intensiteit: De zogenaamde helderheidsinformatie luminantie ‘Y’. De kleurinformatie die zich in de basiskleuren bevindt wordt chrominantie genoemd: ‘C’.

Wanneer nu één basiskleur mist maar wel luminantie aanwezig is kan de missende basiskleur weer uit de aanwezige informatie worden afgeleid, bijvoorbeeld: G = Y – R – B.

In YCbCr zijn de luminantie Y, de blauwe basiskleur Cb en de rode basiskleur Cr aanwezig. Hier is voor gekozen omdat het menselijk oog gevoeliger is voor helderheidsinformatie dan voor kleuren. In YCbCr422 wordt namelijk voor elke pixel de helderheidsinformatie opgeslagen, (bekeken per 4 pixels) maar de beide basiskleuren alleen om de pixel (2 pixels in de bekeken 4 pixels).

In deze standaard wordt de compressie in één richting uitgevoerd. Pixels direct na elkaar (van links naar rechts) zijn zo uitgevoerd. De pixels erboven en eronder staan los van de bekeken ‘rij’. Er zijn ook colorspaces waar de compressie twee dimensionaal is uitgevoerd: zo ontstaat een hogere compressie.

Een backend scaler van een videokaart kan dit soort colorspaces omzetten naar ‘standaard’ RGB. Hierbij kan bij sommige scalers worden gekozen voor interpolatie van de kleurinformatie voor de ‘missende pixels’, terwijl andere scalers alleen pixels kunnen dupliceren.

Referenties:

[1] BeOS API documentation: The Be Book. Included with the BeOS R5 developer tools as HTML documents.
[2] Haiku API documentation: https://www.haiku-os.org/docs/api/
[3] Programmer's Guide to the EGA and VGA Cards, Richard F. Ferraro (Addison-Wesley), ISBN 0-201-12692-3.
[4] The indispensable PC Hardware book, Hans-Peter Messmer (Addison-Wesley), 4e editie, ISBN 0-201-59616-4.
[5] Be Incorporated: http://www.beincorporated.com.
[6] OpenBeOS: http://www.openbeos.org
[7] Haiku: https://www.haiku-os.org/
[8] YellowTab: http://www.yellowtab.com
[9] BeOS R4 Graphics Driver Kit, alpha release 2, 1999-03-30, probably written by Trey Boudreau.
[10] Industry standard 'four character code' colorspaces: http://www.fourcc.org
[11] Author's open BeOS/Haiku Matrox driver: http://rudolfs-place.nl/BeOS/MGAdriver/index.html
[12] Matrox developer relations: register specs for various MGA cards (confidential): http://developer.matrox.com
[13] Author's open BeOS/Haiku Neomagic driver: http://rudolfs-place.nl/BeOS/NMdriver/index.html
[14] Asiliant Technologies: register specs for C&T chipsets for laptops: http://www.asiliant.com/products.htm
[15] Mediaplayer and streaming video solutions. VideoLan/VLC: http://www.videolan.org
[16] VESA organisation's VBE3 open standard: VESA Bios Extensions version 3: http://www.vesa.org/vbelink.html
[17] BeOS R5PE, updates, and developer tools, free for non-commercial use: http://www.bebits.com/app/2680
[18] The premium BeOS software site: http://www.bebits.com
[19] Author's BeTVOut: 'TVout for Nvidia cards with the BeOS': http://betvout.sourceforge.net
[20] FreeBSD http://www.freebsd.org
[21] Linux Xfree86 project sources: http://www.xfree.org
[22] Linux kernel sources: http://www.kernel.org
[23] United States Code van het U.S. house of Representatives: http://www4.law.cornell.edu/uscode/17/1201.html