How to add an (S)VGA driver to XFree86

 Issue 1.3 - May 29, 1994

Contents

1 Introduction

Adding support for a new SVGA chipset to XFree86 is a challenging project for someone who wants to learn more about hardware-level programming. It can be fraught with hazards (in particular, crashing the machine is all too common). But in the end, when the server comes up and functions, it is immensely satisfying.

Adding support for an SVGA chipset does not change any of the basic functioning of the server. It is still a dumb 8-bit PseudoColor server or 1-bit StaticGray server. Adding support for new hardware (e.g. accelerated chips) is a major undertaking, and is not anywhere near formalized enough yet that it can be documented.

Nonetheless, the driver-level programming here is a good introduction. And can well be the first step for adding support for an accelerated chipset, as many are SVGA-supersets. Writing an SVGA-level driver for the chipset can provide a stable development platform for making use of new features (in fact, this has been done for the S3, Cirrus, and WD accelerated chipsets, for internal use as the accelerated servers are developed for XFree86 2.0).

Now let's get down to it. In addition to this documentation, a stub driver has been provided. This should provide you a complete framework for your new driver. Don't let the size of this document persuade you that this is an overly difficult task. A lot of work has been put into making this document as close to complete as possible; hence it should, in theory, be possible to use this as a cookbook, and come out with a working driver when you reach the end. I do advise that you read it all the way through before starting.

2 Getting Started

The first step in developing a new driver is to get the documentation for your chipset. I've included a list of vendor contact information that I have collected so far (it's far from complete, so if you have any that isn't on the list, please send it to me). You need to obtain the databook for the chipset. Make sure that the person you speak to is aware that you intend to do register-level programming (so they don't send you the EE-style datasheet). Ask for any example code, or developer's kits, etc. I've learned that at the SVGA level, in general, a databook that lists and describes the registers is the most you can hope to find.

If you are not familiar with VGA register-level programming, you should get (and read!) a copy of Richard Ferraro's bible (see references below). The best way to understand what is happening in the server is to study the workings of the monochrome server's "generic" server, and compare it with the documentation in Ferraro's book (be aware that there are a few errors in the book). You can find the generic-VGA-register handling functions in the file "vgaHW.c".

Once you understand what's happening in the generic server, you should study one or more of the existing SVGA drivers. Obtain the databook for a supported SVGA chipset, and study the documentation along with the code. When you have a good understanding of what that driver does over and above the generic VGA, you will know what information you need to obtain from the databook for the new chipset. Once you have this information, you are ready to begin work on your new driver.

3 Directory Tree Structure

Here is an outline of the directory tree structure for the source tree. Only directories/files that are relevant to writing a driver are presented. The structure for the Link Kit is presented below.

xc/config/cf/

- **site.def**
  Local configuration customization
- **xf86site.def**
  XFree86 local configuration customization

xc/programs/Xserver/hw/xfree86/

The server source

- **common/**
  Files common to all of the server (XF86Config parser, I/O device handlers, etc)
  - **xf86.h**
    Contains the 'ScrnInfoRec' data structure
  - **xf86_Option.h**
    Contains option flags
  - **compiler.h**
    Contains in-line assembler macros and utility functions
- **os-support/**
  OS-support layer
  - **assyntax.h**
    Contains macro-ized assembler mnemonics
  - **xf86_OSlib.h**
    OS-support includes, defines, and prototypes
- **LinkKit/**
  - **site.def.LK**
    Template for Link Kit site.def

vga256/

256-color VGA server directories

vga/

The generic VGA handling code

vga.h

Contains the 'vgaVideoChipRec' and 'vgaHWRec' data structures

vgaHW.c

Contains the generic-VGA-register handling functions **vgaHWInit()**, **vgaHWSave()** and **vgaHWRestore()**.

drivers/

Contains the SVGA driver subdirectories. Each contains an Imakefile, a .c file for the driver, and a .s file for the bank- switching functions.

vga2/

The monochrome vga server directories. Most of the files are linked from vga256, and the differences handled by conditional compilation.

drivers/

The SVGA driver subdirectories. The 'generic' VGA driver is also located here.

vga16/

The 16-color vga server directories. Most of the files are linked from vga256, and the differences handled by conditional compilation.

drivers/

The SVGA driver subdirectories.

VGADriverDoc/

This documentation and the stub driver.

The Link Kit is usually installed in /usr/X11R6/lib/Server. The Link Kit contains everything that is needed to relink the server. It is possible to write a new driver and build a new server without having even the server source installed.

Server/

site.def

Local configuration customization

include/

All of the include files listed under the 'common' directory above

drivers/

All of the SVGA drivers

vga2/

The SVGA driver subdirectories.

vga16/

The SVGA driver subdirectories.

vga256/
The SVGA driver subdirectories.

VGADriverDoc/
The directory with this documentation and the stub driver. 'vgaHW.c' is also copied here, for reference (it is not built as part of the Link Kit).

4 Setting Up The Build Information

This section describes the peripheral configuration and build steps that must be performed to set up for your new driver. The steps are the same whether you are building from the source tree of from the Link Kit; only the locations of the files is different. Here are the configuration steps that must be followed:

1. Choose the name for your driver subdirectory and data structures. Since the current driver scheme allows (in fact, encourages) putting drivers for multiple related chipsets in a single driver, it is usually best to use the vendor name, rather than a chipset version. The fact that older XFree86 drivers do not follow this convention should not deter you from using it now - most of that code was developed before the driver interface had been made flexible and extensible.

   For this documentation, we'll use chips from the SuperDuper Chips vendor. Hence, we'll use 'sdc' for the name of the driver.

2. Decide whether your driver will support the color server, the monochrome server, or both. For this documentation, we will assume that both the color and monochrome servers will be supported. If you intend to support only the color server, the steps for the monochrome server can be ignored. If you intend to support only the monochrome server, the steps for the color server listed should be performed for the monochrome server, and the monochrome steps ignored. Most of the existing drivers support only the color or both servers; the "generic" driver is the only driver (currently) that supports just the monochrome server.

3. Create your driver directories:

   - If you are working in the source tree, create the following directories:

     ```
     xc/programs/Xserver/hw/xfree86/vga256/drivers/sdc
     xc/programs/Xserver/hw/xfree86/vga16/drivers/sdc
     xc/programs/Xserver/hw/xfree86/vga2/drivers/sdc
     ```

   - If you are working in the Link Kit, create the following directories:

     ```
     /usr/X11R6/lib/Server/drivers/vga256/sdc
     /usr/X11R6/lib/Server/drivers/vga16/sdc
     /usr/X11R6/lib/Server/drivers/vga2/sdc
     ```

4. Set up the Imakefile parameters to cause your driver to be built:

   - If you are working in the source tree:

    (a) Edit the file xc/config/cf/xfree86.cf, and add 'sdc' to the list for the definitions for 'XF86Vga256Drivers', 'XF86Vga16Drivers' and 'XF86Vga2Drivers'. You should put 'sdc' just before 'generic' in the list (i.e. second last), to ensure that none of the other driver's probe functions incorrectly detect the 'sdc' chipset .
    (b) Edit the file xc/config/cf/xf86site.def, and add the same entries in this file (this is just a comment that shows the default values).
    (c) Edit the site.def.LK file in xc/programs/Xserver/hw/xfree86/LinkKit/, and add the same entries in this file. This is the prototype 'site.def' file that will be installed in the Link Kit.

- If you are working in the Link Kit, edit the file /usr/X11R6/lib/Server/site.def, and add 'sdc' to the 'XF86Vga256Drivers', 'XF86Vga16Drivers' and 'XF86Vga2Drivers' definitions as described in (a) above.

5. Now copy the prototype files into your new directories:
   - If you are working in the source tree, copy the 'stub' files as follows (directories are below xc/programs/Xserver):

     Imakefile.stub =>
     hw/xfree86/vga256/drivers/sdc/Imakefile

     stub_driver.c =>
     hw/xfree86/vga256/drivers/sdc/sdc_driver.c

     stub_bank.s =>
     hw/xfree86/vga256/drivers/sdc/sdc_bank.s

     Imakefile.stub =>
     hw/xfree86/vga16/drivers/sdc/Imakefile (then edit this Imakefile and make the changes described in the comments).

     Imakefile.stub =>
     hw/xfree86/vga2/drivers/sdc/Imakefile (then edit this Imakefile and make the changes described in the comments).

   - If you are working in the Link Kit, copy the 'stub' files as follows:

     Imakefile.stub =>
     /usr/X11R6/lib/Server/drivers/vga256/sdc/Imakefile

     stub_driver.c =>
     /usr/X11R6/lib/Server/drivers/vga256/sdc/sdc_driver.c

     stub_bank.s =>
     /usr/X11R6/lib/Server/drivers/vga256/sdc/sdc_bank.s

     Imakefile.stub =>
     /usr/X11R6/lib/Server/drivers/vga16/sdc/Imakefile (then edit this Imakefile and make the changes described in the comments).

     Imakefile.stub =>
     /usr/X11R6/lib/Server/drivers/vga2/sdc/Imakefile (then edit this Imakefile and make the changes described in the comments).

6. Edit each of the files you've just copied, and replace 'stub' with 'sdc' and 'STUB' with 'SDC' wherever they appear.

That's all the prep work needed. Now it's time to work on the actual driver.

5 The Bank-Switching Functions

The normal VGA memory map is 64k starting at address 0xA0000. To access more than 64k of memory, SuperVGA chipsets implement "bank switching" - the high-order address bits are used to select the bank of memory in which operations will take place. The size and number of these banks varies, and will be spelled out in the chipset documentation. A chipset will have zero, one or two bank registers. Likely the ONLY case of zero bank registers is a generic VGA, and hence is not a concern.

Note that some of the newer chipsets (e.g. Trident 8900CL, Cirrus) allow for a linear mapping of the video memory. While using such a scheme would improve the performance of the server, it is not currently supported. Hence there is no way to use such features for a new chipset.

Most SVGA chipsets have two bank registers. This is the most desirable structure (if any banking structure can be called "desirable"), because data can be moved from one area of the screen to another with a simple 'mov' instruction. There are two forms of dual-banking - one where the two bank operations define a read-only bank and a write-only bank, and one with two read/write windows. With the first form, the entire SVGA memory window is used for both read a write operations, and the two bank registers determine which bank is actually used (e.g. ET3000, ET4000). With the second form, the SVGA memory window is split into two read/write banks, with each bank pointer being used to control one window. In this case, one window is used for read operations and the other for write operations (e.g. PVGA1/Western Digital, Cirrus).

A chipset that has a single bank register uses that one bank for both read and write access. This is problematic, because copying information from one part of the screen to another requires that the data be read in, stored, and then written out. Fortunately, the server is able to handle both one-bank and two-bank chipsets; the determination of behavior is defined by an entry in the driver data structure described below.

A driver requires that three assembly-language functions be written, in the file '`sdc_bank.s`'. These functions set the read bank - **SDCSetRead()**, the write bank - **SDCSetWrite()**, and set both banks - **SDCSetReadWrite()**. For a chipset with only one bank, all three will be declared as entry points to the same function (see the "tvga8900" driver for an example).

The functions are fairly simple - the bank number is passed to the function in register %al. The function will shift, bitmask, etc - whatever is required to put the bank number into the correct form - and then write it to the correct I/O port. For chipsets where the two banks are read-only HERE and write-only, the **SetReadWrite()** function will have to do this twice - once for each bank. For chipsets with two independent read/write windows, the **SetReadWrite()** function should use the same bank as the **SetWrite()** function.

A special note - these functions MUST be written in the macroized assembler format defined in the header file "assyntax.h". This will ensure that the correct assembler code will be generated, regardless of OS. This macroized format currently supports USL, GNU, and Intel assembler formats.

That's all there is to the banking functions. Usually the chipset reference will give examples of this code; if not, it is not difficult to figure out, especially using the other drivers as examples.

6 The Driver Itself

Now it's time to get down to the real work - writing the major driver functions in the files sdc_driver.c. First, an overview of what the responsibilities of the driver are:

1. Provide a chipset-descriptor data structure to the server. This data structure contains pointers to the driver functions and some data-structure initialization as well.
2. Provide a driver-local data structure to hold the contents of the chipset registers. This data structure will contain a generic part and a driver-specific part. It is used to save the initial chipset state, and is initialized by the driver to put the chipset into different modes.
3. Provide an identification function that the server will call to list the chipsets that the driver is capable of supporting.
4. Provide a probe function that will identify this chipset as different from all others, and return a positive response if the chipset this driver supports is installed, and a negative response otherwise.
5. Provide a function to select dot-clocks available on the board.
6. Provide functions to save, restore, and initialize the driver- local data structure.
7. Provide a function to set the starting address for display in the video memory. This implements the virtual-screen for the server.
8. Perhaps provide a function for use during VT-switching.
9. Perhaps provide a function to check if each mode is suitable for the chipset being used.

Before stepping through the driver file in detail, here are some important issues:

1. If your driver supports both the color and monochrome servers, you should take care of both cases in the same file. Most things are the same - you can differentiate between the two with the MONOVGA `#define`. If the 16 color server is supported, code specific to it can be enabled with the XF86VGA16 `#define`. In most cases it is sufficient to put the following near the top of the stub_driver.c file:

   ```
   #ifdef XF86VGA16
   #define MONOVGA
   #endif
   ```

2. The color server uses the SVGA's 8-bit packed-pixel mode. The monochrome and vga16 servers uses the VGA's 16-color mode (4 bit-planes). Only one plane is enabled for the monochrome server.
3. It is possible for you to define your monochrome driver so that no bank-switching is done. This is not particularly desirable, as it yields only 64k of viewing area.

Keeping these things in mind, you need to find the registers from your SVGA chipset that control the desired features. In particular, registers that control:

1. Clock select bits. The two low-order bits are part of the standard Miscellaneous Output Register; most SVGA chipsets will include 1 or 2 more bits, allowing the use of 8 or 16 discrete clocks.

2. Bank selection. The SVGA chipset will have one or two registers that control read/write bank selection.

3. CRTC extensions. The standard VGA registers don't have enough bits to address large displays. So the SVGA chipsets have extension bits.

4. Interlaced mode. Standard VGA does not support interlaced displays. So the SVGA chipset will have a bit somewhere to control interlaced mode. Some chipsets require additional registers to be set up to control interlaced mode

5. Starting address. The standard VGA only has 16 bits in which to specify the starting address for the display. This restricts the screen size usable by the virtual screen feature. The SVGA chipset will usually provide one or more extension bits.

6. Lock registers. Many SVGA chipset prevent modification of extended registers unless the registers are first "unlocked". You will need to disable protection of any registers you will need for other purposes.

7. Any other facilities. Some chipset may, for example, require that certain bits be set before you can access extended VGA memory (beyond the IBM-standard 256k). Or other facilities; read through all of the extended register descriptions and see if anything important leaps out at you.

If you are fortunate, the chipset vendor will include in the databook some tables of register settings for various BIOS modes. You can learn a lot about what manipulations you must do by looking at the various BIOS modes.

6.1 Multiple Chipsets And Options

It is possible, and in fact desirable, to have a single driver support multiple chipsets from the same vendor. If there are multiple supported chipsets, then you would have a series of #define's for them, and a variable 'SDCchipset', which would be used throughout the driver when distinctions must be made. See the Trident and PVGA1/WD drivers for examples (the Tseng ET3000 and ET4000 are counter-examples - these were implemented before the driver interface allowed for multiple chipsets, so this example should NOT be followed). Note that you should only distinguish versions when your driver needs to do things differently for them. For example, suppose the SDC driver supports the SDC-1a, SDC-1b, and SDC-2 chipsets. The -1a and -1b are essentially the same, but different from the -2 chipset. Your driver should support the -1 and -2 chipsets, and not distinguish between the -1a and -1a. This will simplify things for the end user.

In cases where you want to give the user control of driver behavior, or there are things that cannot be determined without user intervention, you should use "option" flags. Say that board vendors that use the SDC chipsets have the option of providing 8 or 16 clocks. There's no way you can determine this from the chipset probe, so you provide an option flag to let the user select the behavior from the XF86Config file. The option flags are defined in the file "xf86_option.h". You should look to see if there is already a flag that can be reused. If so, use it in your driver. If not, add a new #define, and define the string-¿symbol mapping in the table in that file. To see how option flags are used, look at the ET4000, PVGA1/WD, and Trident drivers.

6.2 Data Structures

Once you have an understanding of what is needed from the above description, it is time to fill in the driver data structures. First we will deal with the 'vgaSDCRec' structure. This data structure is the driver-local structure that holds the SVGA state information. The first entry in this data structure is ALWAYS 'vgaHWRec std'. This piece holds the generic VGA portion of the information. After that, you will have one 'unsigned char' field for each register that will be manipulated by your driver. That's all there is to this data structure.

Next you must initialize the 'SDC' structure (type 'vgaVideoChipRec'). This is the global structure that identifies your driver to the server. Its name MUST be 'SDC', in all caps - i.e. it must match the directory name for your driver. This is required so that the Link Kit reconfiguration can identify all of the requisite directories and global data structures.

The first section of this structure simply holds pointers to the driver functions.

Next, you must initialize the information about how your chipset does bank switching. The following fields must be filled in:

1. ChipMapSize - the amount of memory that must be mapped into the server's address space. This is almost always 64k (from 0xA0000 to 0xAFFFF). Some chipsets use a 128k map (from 0xA0000 to 0xBFFFF). If your chipset gives an option, use the 64k window, as a 128k window rules out using a Hercules or Monochrome Display Adapter card with the SVGA.

2. ChipSegmentSize - the size of each bank within the ChipMapSize window. This is usually also 64k, however, some chipsets split the mapped window into a read portion and a write portion (for example the PVGA1/Western Digital chipsets).

3. ChipSegmentShift - the number of bits by which an address will be shifted right to mask of the bank number. This is log-base-2 of ChipSegmentSize.

4. ChipSegmentMask - a bitmask used to mask off the address within a given bank. This is (ChipSegmentSize-1).

5. ChipReadBottom,ChipReadTop - the addresses within the mapped window in which read operations can be done. Usually 0, and 64k, respectively, except for those chipset that have separate read and write windows.

6. ChipWriteBottom,ChipWriteTop - same as above, for write operations.

7. ChipUse2Banks - a boolean value for whether this chipset has one or two bank registers. This is used to set up the screen-to-screen operations properly.

There are three more fields that must be filled in:

1. ChipInterlaceType - this is either VGA_NO_DIVIDE_VERT or VGA_DIVIDE_VERT. Some chipsets require that the vertical timing numbers be divided in half for interlaced modes. Setting this flag will take care of that.

2. ChipOptionFlags - this should always be '{0,}' in the data structure initialization. This is a bitfield that contains the Option flags that are valid for this driver. The appropriate bits are initialized at the end of the Probe function.

3. ChipRounding - this gets set to the multiple by which the virtual width of the display must be rounded for the 256-color server. This value is usually 8, but may be 4 or 16 for some chipsets.

6.3 The Ident() function

The **Ident()** function is a very simple function. The server will call this function repeatedly, until a NULL is returned, when printing out the list of configured drivers. The **Ident()** function should return a chipset name for a supported chipset. The function is passed a number which increments from 0 on each iteration.

6.4 The ClockSelect() function

The **ClockSelect()** function is used during clock probing (i.e. when no 'Clocks' line is specified in the XF86Config file) to select the dot-clock indicated by the number passed in the parameter. The function should set the chipset's clock-select bits according to the passed-in number. Two dummy values will be passed in as well (CLK_REG_SAVE, CLK_SAVE_RESTORE). When CLK_REG_SAVE is passed, the function should save away copies of any registers that will be modified during clock selection. When CLK_REG_RESTORE is passed, the function should restore these registers. This ensure that the clock-probing cannot corrupt registers.

This function should return FALSE if the passed-in index value is invalid or if the clock can't be set for some reason.

6.5 The Probe() function

The **Probe()** function is perhaps the most important, and perhaps the least intuitive function in the driver. The Probe function is required to identify the chipset independent of all other chipsets. If the user has specified a '`Chipset`' line in the XF86Config file, this is a simple string comparison check. Otherwise, you must use some other technique to figure out what chipset is installed. If you are lucky, the chipset will have an identification mechanism (ident/version registers, etc), and this will be documented in the databook. Otherwise, you will have to determine some scheme, using the reference materials listed below.

The identification is often done by looking for particular patterns in register, or for the existence of certain extended registers. Or with some boards/chipsets, the requisite information can be obtained by reading the BIOS for certain signature strings. The best advise is to study the existing probe functions, and use the reference documentation. You must be certain that your probe is non-destructive - if you modify a register, it must be saved before, and restored after.

Once the chipset is successfully identified, the **Probe()** function must do some other initializations:

1. If the user has not specified the '`VideoRam`' parameter in the XF86Config file, the amount of installed memory must be determined.

2. If the user has not specified the '`Clocks`' parameter in the XF86Config file, the values for the available dot-clocks must be determined. This is done by calling the **vgaGetClocks()** function, and passing it the number of clocks available and a pointer to the **ClockSelect()** function.

3. It is recommended that the 'maxClock' field of the server's 'vga256InfoRec' structure be filled in with the maximum dot-clock rate allowed for this chipset (specified in KHz). If this is not filled in a probe time, a default (currently 90MHz) will be used.

4. The 'chipset' field of the server's 'vga256InfoRec' structure must be initialized to the name of the installed chipset.

5. If the driver will be used with the monochrome server, the 'bankedMono' field of the server's 'vga256InfoRec' structure must be set to indicate whether the monochrome driver supports banking.

6. If any option flags are used by this driver, the 'ChipOptionFlags' structure in the 'vgaVideoChipRec' must be initialized with the allowed option flags using the **OFLG_SET()** macro.

6.6 The EnterLeave() function

The **EnterLeave()** function is called whenever the virtual console on which the server runs is entered or left (for OSs without virtual consoles, the function is called when the server starts and again when it exits). The purpose of this function is to enable and disable I/O permissions (for OSs where such is required), and to unlock and relock access to "protected" registers that the driver must manipulate. It is a fairly trivial function, and can be implemented by following the comments in the stub driver.

6.7 The Restore() function

The **Restore()** function is used for restoring a saved video state. Note that 'restore' is a bit of a misnomer - this function is used to both restore a saved state and to install a new one created by the server. The **Restore()** function must complete the following actions:

1. Ensure that Bank 0 is selected, and that any other state information required prior to writing out a new state has been set up.

2. Call **vgaHWRestore()** to restore the generic VGA portion of the state information. This function is in the vgaHW.c file.

3. Restore the chipset-specific portion of the state information. This may be done by simply writing out the register, or by doing a read/modify/write cycle if only certain bits are to be modified. Be sure to note the comment in the sample driver about how to handle clock-select bits.

6.8 The Save() function

The **Save()** function is used to extract the initial video state information when the server starts. The **Save()** function must complete the following actions:

1. Ensure that Bank 0 is selected.
2. Call **vgaHWSave()** to extract the generic VGA portion of the state information. This function is in the vgaHW.c file.
3. Extract the chipset-specific portion of the state information.

6.9 The Init() function

The **Init()** function is the second most important function in the driver (after the **Probe()** function). It is used to initialize a data structure for each of the defined display modes in the server. This function is required to initialize the entire 'vgaSDCRec' data structure with the information needed to put the SVGA chipset into the required state. The generic VGA portion of the structure is initialized with a call to **vgaHWInit()** (also located in vgaHW.c).

Once the generic portion is initialized, the **Init()** function can override any of the generic register initialization, if necessary. All of the other fields are filled in with the correct initialization. The information about the particular mode being initialized is passed in the 'mode' parameter, a pointer to a 'DisplayModeRec' structure. This can be dereferenced to determine the needed parameters.

If you only know how to initialize certain bits of the register, do that here, and make sure that the **Restore()** function does a read/modify/write to only manipulate those bits. Again, refer to the existing drivers for examples of what happens in this function.

6.10 The Adjust() function

The **Adjust()** function is another fairly basic function. It is called whenever the server needs to adjust the start of the displayed part of the video memory, due to scrolling of the virtual screen or when changing the displayed resolution. All it does is set the starting address on the chipset to match the specified coordinate. Follow the comments in the stub driver for details on how to implement it.

6.11 The ValidMode() function

The **ValidMode()** function is required. It is used to check for any chipset-dependent reasons why a graphics mode might not be valid. It gets called by higher levels of the code after the Probe() stage. In many cases no special checking will be required and this function will simply return TRUE always.

6.12 The SaveScreen() function

The **SaveScreen()** function is not needed by most chipsets. This function would only be required if the extended registers that your driver needs will be modified when a synchronous reset is performed on the SVGA chipset (your databook should tell you this). If you do NOT need this function, simply don't define it, and put 'NoopDDA' in its place in the vgaVideoChipRec structure initialization (NoopDDA is a generic-use empty function).

If you DO need this function, it is fairly simple to do. It will be called twice - once before the reset, and again after. It will be passed a parameter of SS_START in the former case, and SS_FINISH in the latter. All that needs to be done is to save any registers that will be affected by the reset into static variables on the SS_START call, and then restore them on the SS_FINISH call.

6.13 The GetMode() function

The **GetMode()** function is not used as of XFree86 1.3; its place in the vgaVideoChipRec should be initialized to 'NoopDDA'.

At some point in the future, this function will be used to enable the server and/or a standalone program using the server's driver libraries to do interactive video mode adjustments. This function will read the SVGA registers and fill in a DisplayModeRec structure with the current video mode.

6.14 The FbInit() function

The **FbInit()** function is required for drivers with accelerated graphics support. It is used to replace default cfb.banked functions with accelerated chip-specific versions. vga256LowlevFuncs is a struct containing a list of functions which can be replaced. This struct defined in vga256.h. Examples of **FbInit()** functions can be found in the et4000, pvga1 and cirrus drivers.

If you do NOT need this function, simply don't define it, and put 'NoopDDA' in its place in the vgaVideoChipRec structure initialization.

7 Building The New Server

As in the setup work, the steps for building the server depend whether you are working in the source tree or in the Link Kit. Here are the steps for the initial build after installing your new driver files:

- If you are working in the source tree, follow these steps:

  Go to xc/programs/Xserver, and enter '`make Makefile`', then '`make Makefiles depend all`'
- If you are working in the Link Kit, follow these steps:
  1. Go to /usr/X11R6/lib/Server, and enter '`./mkmf`'
  2. In the same directory, enter '`make`'

To rebuild the server after the initial build (e.g. after making changes to your driver):

- If you are working in the source tree, follow these steps:
  1. Go to the appropriate drivers/ directory (e.g., xc/programs/Xserver/hw/xfree86/vga256/drivers), and enter '`make`'.

   2. Go to xc/programs/Xserver, and enter '`make loadXF86_SVGA`' (to link the color server), '`make loadXF86_VGA16`' (to link the 16 color server) or '`make loadXF86_Mono`' (to link the mono server).

- If you are working in the Link Kit, follow these steps:

   1. Go to the appropriate driver directory, and enter '`make`'.
   2. Go to /usr/X11R6/lib/server, and enter '`make loadXF86_SVGA`' (to link the color server) or '`make loadXF86_VGA16`' (to link the 16 color server) or '`make loadXF86_Mono`' (to link the mono server).

8 Debugging

Debugging a new driver can be a painful experience, unfortunately. It is likely that incorrect programming of the SVGA chipset can lock up your machine. More likely, however, is that the display will be lost, potentially requiring a reboot to correct. It is HIGHLY recommended that the server be run from an attached terminal or a network login. This is the only rational way in which a debugger can be used on the server. Attempting to use multiple VTs for debugging is basically a waste of time.

Because of the potential for locking up the machine, it is a VERY good idea to remember to do a 'sync' or two before starting the server. In addition, any unnecessary filesystems should be unmounted while the debugging session is going on (to avoid having to run unnecessary fsck's).

By default the server is built without debugging symbols. The server can grow VERY large with debugging enabled. It is very simple to rebuild your driver for debugging, though. Do the following:

1. Go to the driver directory.
2. Edit the Makefile. Look for the SECOND definition of '`CDEBUGFLAGS`'. Change this definition to

   ```
   CDEBUGFLAGS = -g -DNO_INLINE
   ```

   (this will enable debugging symbols and disable inlining of functions, which can make single-stepping a nightmare).
3. Remove the 'sdc_driver.o' file.
4. Now follow the steps above for rebuilding the server.

   (Alternatively, instead of editing the Makefile, you can simply do '`make CDEBUGFLAGS="-g -DNO_INLINE"`' after removing the old .o file, then rebuild the server as described above).

This will give you a server with which you can set breakpoints in the driver functions and single-step them. If you are working in the source tree, and just learning about SVGA programming, it may be useful to rebuild vgaHW.c with debugging as well.

9 Advice

I cannot stress this enough - study all available references, and the existing code, until you understand what is happening. Do this BEFORE you begin writing a driver. This will save you a massive amount of headache. Try to find a driver for a chipset that is similar to yours, if possible. Use this as an example, and perhaps derive your driver from it.

Do not let the gloom-and-doom in the debugging section discourage you. While you will probably have problems initially (I still do), careful, deliberate debugging steps can bear fruit very quickly. It is likely that, given a good understanding of the chipset, a driver can be written and debugged in a day or two. For someone just learning about this kind of programming, a week is more reasonable.

10 Advanced Topics

Newer chipsets are getting into two advanced areas: programmable clock generators, and accelerated capabilities (BitBlt, line drawing, HW cursor). These are new areas, and the formal interfaces to them are not yet defined. It is advised that you contact the XFree86 team and get involved with the development/beta-testing team if you need to be working in these areas.

11 References

- Programmer's Guide to the EGA and VGA Cards, 2nd ed.
  Richard Ferraro
  Addison-Wesley, 1990
  ISBN 0-201-57025-4
  (This is the bible of SVGA programming - it has a few errors, so watch out).

- vgadoc3.zip
  Finn Thoegersen
  (This is a collection of SVGA and other chipset documentation. It is available on most MS-DOS/Windows related FTP archives, including wuarchive. It is DOS/BIOS oriented, but is still extremely useful, especially for developing probe functions).

12 Vendor Contact Information

ATI Technologies (VGA-Wonder, Mach8, Mach32) 33 Commerce Valley Drive East

Thornhill, Ontario
Canada L3T 7N6
(905) 882-2600 (sales)
(905) 882-2626 (tech support)
(905) 764-9404 (BBS)

(905) 882-0546 (fax)

Chips & Technologies

???

Cirrus Logic (SVGA, Accelerators - CL-GD5426)

3100 West Warren Ave.
Fremont, CA 94538
(510) 623-8300 (N. CA, USA)
(49) 8152-40084 (Germany)
(44) 0727-872424 (UK)

Genoa Systems (GVGA)

75 E. Trimble Road
San Jose, CA 95131
(408) 432-9090 (sales)
(408) 432-8324 (tech support)

Headland Technologies, Inc (Video-7 VGA 1024i, VRAM II)

46221 Landing Parkway
Fremont, CA 94538
(415) 623-7857

Oak Technology, Inc (OTI-067,OTI-077)

139 Kifer Ct.
Sunnyvale, CA 94086
(408) 737-0888
(408) 737-3838 (fax)

S3 (911, 924, 801

805, 928)/ (408) 980-5400

Trident Microsystems Inc (8800, 8900, 9000)

205 Ravendale Dr
Mountainside, CA 94043
(415) 691-9211

Tseng Labs Inc,

6 Terry Drive
Newtown, PA 18940
(215) 968-0502

Weitek (Power9000, 5186)

1060 E. Arques Ave,
Sunnyvale, CA 94086
(408) 738-5765

Western Digital

(714) 932-4900

`$XConsortium: VGADriv.sgml,v 1.3 95/01/23 15:34:51 kaleb Exp $`
`$XFree86: xc/programs/Xserver/hw/xfree86/doc/sgml/VGADriv.sgml,v 3.5 1995/01/28 16:02:34 dawes Exp $`