XFree86 server 4.x Design (DRAFT)

The XFree86 Project, Inc

19 December 2003

NOTE: This is a DRAFT document, and the interfaces described here are subject to change without notice.

1. Preface

The broad design principles are:

- keep it reasonable
  - We cannot rewrite the complete server
  - We don't want to re-invent the wheel
- keep it modular
  - As many things as possible should go into modules
  - The basic loader binary should be minimal
  - A clean design with well defined layering is important
  - DDX specific global variables are a nono
  - The structure should be flexible enough to allow future extensions
  - The structure should minimize duplication of common code
- keep important features in mind
  - multiple screens, including multiple instances of drivers
  - mixing different color depths and visuals on different and ideally even on the same screen
  - better control of the PCI device used
  - better config file parser
  - get rid of all VGA compatibility assumptions

Unless we find major deficiencies in the DIX layer, we should avoid making changes there.

2. The XF86Config File

The XF86Config file format is similar to the old format, with the following changes:

2.1 Device section

The **Device** sections are similar to what they used to be, and describe hardware-specific information for a single video card. **Device** Some new keywords are added:

Driver "drivername"
: Specifies the name of the driver to be used for the card. This is mandatory.

BusID "busslot"
: Specifies uniquely the location of the card on the bus. The purpose is to identify particular cards in a multi-headed configuration. The format of the argument is intentionally vague, and may be architecture dependent. For a PCI bus, it is something like "bus:slot:func".

A **Device** section is considered "active" if there is a reference to it in an active **Screen** section.

2.2 Screen section

The **Screen** sections are similar to what they used to be. They no longer have a **Driver** keyword, but an **Identifier** keyword is added. (The **Driver** keyword may be accepted in place of the **Identifier** keyword for compatibility purposes.) The identifier can be used to identify which screen is to be active when multiple **Screen** sections are present. It is possible to specify the active screen from the command line. A default is chosen in the absence of one being specified. A **Screen** section is considered "active" if there is a reference to it either from the command line, or from an active **ServerLayout** section.

2.3 InputDevice section

The **InputDevice** section is a new section that describes configuration information for input devices. It replaces the old **Keyboard**, **Pointer** and **XInput** sections. Like the **Device** section, it has two mandatory keywords: **Identifier** and **Driver**. For compatibility purposes the old **Keyboard** and **Pointer** sections are converted by the parser into **InputDevice** sections as follows:

Keyboard
: **Identifier** "Implicit Core Keyboard"

  Driver "keyboard"

Pointer
: **Identifier** "Implicit Core Pointer"

  Driver "mouse"

An **InputDevice** section is considered active if there is a reference to it in an active **ServerLayout** section. An **InputDevice** section may also be referenced implicitly if there is no **ServerLayout** section, if the `-screen` command line options is used, or if the **ServerLayout** section doesn't reference any **InputDevice** sections. In this case, the first sections with drivers "keyboard" and "mouse" are used as the core keyboard and pointer respectively.

2.4 ServerLayout section

The **ServerLayout** section is a new section that is used to identify which **Screen** sections are to be used in a multi-headed configuration, and the relative layout of those screens. It also identifies which **InputDevice** sections are to be used. Each **ServerLayout** section has an identifier, a list of **Screen** section identifiers, and a list of **InputDevice** section identifiers. **ServerFlags** options may also be included in a **ServerLayout** section, making it possible to override the global values in the **ServerFlags** section.

A **ServerLayout** section can be made active by being referenced on the command line. In the absence of this, a default will be chosen (the first one found). The screen names may optionally be followed by a number specifying the preferred screen number, and optionally by information specifying the physical positioning of the screen, either in absolute terms or relative to another screen (or screens). When no screen number is specified, they are numbered according to the order in which they are listed. The old (now obsolete) method of providing the positioning information is to give the names of the four adjacent screens. The order of these is top, bottom, left, right. Here is an example of a **ServerLayout** section for two screens using the old method, with the second located to the right of the first:

```
Section "ServerLayout"
  Identifier "Main Layout"
  Screen     0 "Screen 1" ""  ""  ""  "Screen 2"
  Screen     1 "Screen 2"
  Screen     "Screen 3"
EndSection
```

The preferred way of specifying the layout is to explicitly specify the screen's location in absolute terms or relative to another screen.

In the absolute case, the upper left corner's coordinates are given after the **Absolute** keyword. If the coordinates are omitted, a value of `(0,0)` is assumed. An example of absolute positioning follows:

```
Section "ServerLayout"
  Identifier "Main Layout"
  Screen     0 "Screen 1" Absolute 0 0
  Screen     1 "Screen 2" Absolute 1024 0
  Screen     "Screen 3" Absolute 2048 0
EndSection
```

In the relative case, the position is specified by either using one of the following keywords followed by the name of the reference screen:

RightOf

LeftOf

Above

Below

Relative

When the **Relative** keyword is used, the reference screen name is followed by the coordinates of the new screen's origin relative to reference screen. The following example shows how to use some of the relative positioning options.

```
Section "ServerLayout"
  Identifier "Main Layout"
  Screen     0 "Screen 1"
  Screen     1 "Screen 2" RightOf "Screen 1"
  Screen     "Screen 3" Relative "Screen 1" 2048 0
EndSection
```

2.5 Options

Options are used more extensively. They may appear in most sections now. Options related to drivers can be present in the **Screen**, **Device** and **Monitor** sections and the **Display** subsections. The order of precedence is **Display**, **Screen**, **Monitor**, **Device**. Options have been extended to allow an optional value to be specified in addition to the option name. For more details about options, see the *Options* (section 10., page 33) section for details.

3. Driver Interface

The driver interface consists of a minimal set of entry points that are required based on the external events that the driver must react to. No non-essential structure is imposed on the way they are used beyond that. This is a significant difference compared with the old design.

The entry points for drawing operations are already taken care of by the framebuffer code (including, XAA). Extensions and enhancements to framebuffer code are outside the scope of this document.

This approach to the driver interface provides good flexibility, but does increase the complexity of drivers. To help address this, the XFree86 common layer provides a set of "helper" functions to take care of things that most drivers need. These helpers help minimise the amount of code duplication between drivers. The use of helper functions by drivers is however optional, though encouraged. The basic philosophy behind the helper functions is that they should be useful to many drivers, that they should balance this against the complexity of their interface. It is inevitable that some drivers may find some helpers unsuitable and need to provide their own code.

Events that a driver needs to react to are:

ScreenInit
: An initialisation function is called from the DIX layer for each screen at the start of each server generation.

Enter VT
: The server takes control of the console.

Leave VT
: The server releases control of the console.

Mode Switch
: Change video mode.

ViewPort change
: Change the origin of the physical view port.

ScreenSaver state change
: Screen saver activation/deactivation.

CloseScreen
: A close screen function is called from the DIX layer for each screen at the end of each server generation.

In addition to these events, the following functions are required by the XFree86 common layer:

Identify
: Print a driver identifying message.

Probe
: This is how a driver identifies if there is any hardware present that it knows how to drive.

PreInit
: Process information from the XF86Config file, determine the full characteristics of the hardware, and determine if a valid configuration is present.

The VidMode extension also requires:

ValidMode
: Identify if a new mode is usable with the current configuration. The PreInit function (and/or helpers it calls) may also make use of the ValidMode function or

something similar.

Other extensions may require other entry points. The drivers will inform the common layer of these in such cases.

4. Resource Access Control Introduction

Graphics devices are accessed through ranges in I/O or memory space. While most modern graphics devices allow relocation of such ranges many of them still require the use of well established interfaces such as VGA memory and IO ranges or 8514/A IO ranges. With modern buses (like PCI) it is possible for multiple video devices to share access to these resources. The RAC (Resource Access Control) subsystem provides a mechanism for this.

4.1 Terms and Definitions

4.1.1 Bus

''Bus'' is ambiguous as it is used for different things: it may refer to physical incompatible extension connectors in a computer system. The RAC system knows two such systems: The ISA bus and the PCI bus. (On the software level EISA, MCA and VL buses are currently treated like ISA buses). ''Bus'' may also refer to logically different entities on a single bus system which are connected via bridges. A PCI system may have several distinct PCI buses connecting each other by PCI-PCI bridges or to the host CPU by HOST-PCI bridges.

Systems that host more than one bus system link these together using bridges. Bridges are a concern to RAC as they might block or pass specific resources. PCI-PCI bridges may be set up to pass VGA resources to the secondary bus. PCI-ISA buses pass any resources not decoded on the primary PCI bus to the ISA bus. This way VGA resources (although exclusive on the ISA bus) can be shared by ISA and PCI cards. Currently HOST-PCI bridges are not yet handled by RAC as they require specific drivers.

4.1.2 Entity

The smallest independently addressable unit on a system bus is referred to as an entity. So far we know ISA and PCI entities. PCI entities can be located on the PCI bus by an unique ID consisting of the bus, card and function number.

4.1.3 Resource

''Resource'' refers to a range of memory or I/O addresses an entity can decode.

If a device is capable of disabling this decoding the resource is called sharable. For PCI devices a generic method is provided to control resource decoding. Other devices will have to provide a device specific function to control decoding.

If the entity is capable of decoding this range at a different location this resource is considered relocatable.

Resources which start at a specific address and occupy a single continuous range are called block resources.

Alternatively resource addresses can be decoded in a way that they satisfy the conditions:

```
address & mask == base
```

and

```
base & mask == base
```

Resources addressed in such a way are called sparse resources.

4.1.4 Server States

The resource access control system knows two server states: the SETUP and the OPERATING state. The SETUP state is entered whenever a mode change takes place or the server exits or does VT switching. During this state all entity resources are under resource access control. During OPERATING state only those entities are controlled which actually have shared resources that conflict with others.

5. Control Flow in the Server and Mandatory Driver Functions

At the start of each server generation, `main()` (`dix/main.c`) calls the DDX function `InitOutput()`. This is the first place that the DDX gets control. `InitOutput()` is expected to fill in the global `screenInfo` struct, and one `screenInfo.screen[]` entry for each screen present. Here is what `InitOutput()` does:

5.1 Parse the XF86Config file

This is done at the start of the first server generation only.

The XF86Config file is read in full, and the resulting information stored in data structures. None of the parsed information is processed at this point. The parser data structures are opaque to the video drivers and to most of the common layer code.

The entire file is parsed first to remove any section ordering requirements.

5.2 Initial processing of parsed information and command line options

This is done at the start of the first server generation only.

The initial processing is to determine paths like the **ModulePath**, etc, and to determine which **ServerLayout**, **Screen** and **Device** sections are active.

5.3 Enable port I/O access

Port I/O access is controlled from the XFree86 common layer, and is ''all or nothing''. It is enabled prior to calling driver probes, at the start of subsequent server generations, and when VT switching back to the Xserver. It is disabled at the end of server generations, and when VT switching away from the Xserver.

The implementation details of this may vary on different platforms.

5.4 General bus probe

This is done at the start of the first server generation only.

In the case of ix86 machines, this will be a general PCI probe. The full information obtained here will be available to the drivers. This information persists for the life of the Xserver. In the PCI case, the PCI information for all video cards found is available by calling `xf86GetPciVideoInfo()`.

`pciVideoPtr *xf86GetPciVideoInfo(void)`

> returns a pointer to a list of pointers to `pciVideoRec` entries, of which there is one for each detected PCI video card. The list is terminated with a `NULL` pointer. If no PCI video cards were detected, the return value is `NULL`.

After the bus probe, the resource broker is initialised.

5.5 Load initial set of modules

This is done at the start of the first server generation only.

The core server contains a list of mandatory modules. These are loaded first. Currently the only module on this list is the bitmap font module.

The next set of modules loaded are those specified explicitly in the **Module** section of the config file.

The final set of initial modules are the driver modules referenced by the active **Device** and **InputDevice** sections in the config file. Each of these modules is loaded exactly once.

5.6 Register Video and Input Drivers

This is done at the start of the first server generation only.

When a driver module is loaded, the loader calls its `Setup` function. For video drivers, this function calls `xf86AddDriver()` to register the driver's `DriverRec`, which contains a small set of essential details and driver entry points required during the early phase of `InitOutput()`. `xf86AddDriver()` adds it to the global `xf86DriverList[]` array.

The `DriverRec` contains the driver canonical name, the `Identify()`, `Probe()` and `AvailableOptions()` function entry points as well as a pointer to the driver's module (as returned from the loader when the driver was loaded) and a reference count which keeps track of how many screens are using the driver. The entry driver entry points are those required prior to the driver allocating and filling in its `ScrnInfoRec`.

For a static server, the `xf86DriverList[]` array is initialised at build time, and the loading of modules is not done.

A similar procedure is used for input drivers. The input driver's `Setup` function calls `xf86AddInputDriver()` to register the driver's `InputDriverRec`, which contains a small set of essential details and driver entry points required during the early phase of `InitInput()`. `xf86AddInputDriver()` adds it to the global `xf86InputDriverList[]` array. For a static server, the `xf86InputDriverList[]` array is initialised at build time.

Both the `xf86DriverList[]` and `xf86InputDriverList[]` arrays have been initialised by the end of this stage.

Once all the drivers are registered, their `ChipIdentify()` functions are called.

`void ChipIdentify(int flags)`

> This is expected to print a message indicating the driver name, a short summary of what it supports, and a list of the chipset names that it supports. It may use the xf86PrintChipsets() helper to do this.

```
void xf86PrintChipsets(const char *drvname, const char *drvmsg,
        SymTabPtr chips)
```

This function provides an easy way for a driver's ChipIdentify function to format the identification message.

5.7 Initialise Access Control

This is done at the start of the first server generation only.

The Resource Access Control (RAC) subsystem is initialised before calling any driver functions that may access hardware. All generic bus information is probed and saved (for restoration later). All (shared resource) video devices are disabled at the generic bus level, and a probe is done to find the ''primary'' video device. These devices remain disabled for the next step.

5.8 Video Driver Probe

This is done at the start of the first server generation only. The `ChipProbe()` function of each registered video driver is called.

```
Bool ChipProbe(DriverPtr drv, int flags)
```

The purpose of this is to identify all instances of hardware supported by the driver. The flags value is currently either 0, `PROBE_DEFAULT` or `PROBE_DETECT`. `PROBE_DETECT` is used if "-configure" or "-probe" command line arguments are given and indicates to the `Probe()` function that it should not configure the bus entities and that no XF86Config information is available.

The probe must find the active device sections that match the driver by calling `xf86MatchDevice()`. The number of matches found limits the maximum number of instances for this driver. If no matches are found, the function should return `FALSE` immediately.

Devices that cannot be identified by using device-independent methods should be probed at this stage (keeping in mind that access to all resources that can be disabled in a device-independent way are disabled during this phase). The probe must be a minimal probe. It should just determine if there is a card present that the driver can drive. It should use the least intrusive probe methods possible. It must not do anything that is not essential, like probing for other details such as the amount of memory installed, etc. It is recommended that the `xf86MatchPciInstances()` helper function be used for identifying matching PCI devices, and similarly the `xf86MatchIsaInstances()` for ISA (non-PCI) devices (see the *RAC* (section 9., page 22) section). These helpers also checks and claims the appropriate entity. When not using the helper, that should be done with `xf86CheckPciSlot()` and `xf86ClaimPciSlot()` for PCI devices and `xf86ClaimIsaSlot()` for ISA devices (see the *RAC* (section 9., page 22) section).

The probe must register all non-relocatable resources at this stage. If a resource conflict is found between exclusive resources the driver will fail immediately. This is usually best done with the `xf86ConfigPciEntity()` helper function for PCI and `xf86ConfigIsaEntity()` for ISA (see the *RAC* (section 9., page 22) section). It is possible to register some entity specific functions with those helpers. When not using the helpers, the `xf86AddEntityToScreen()` `xf86ClaimFixedResources()` and `xf86SetEntityFuncs()` should be used instead (see the *RAC* (section 9., page 22) section).

If a chipset is specified in an active device section which the driver considers relevant (ie it has no driver specified, or the driver specified matches the driver doing the probe), the Probe must return `FALSE` if the chipset doesn't match one supported by the driver.

If there are no active device sections that the driver considers relevant, it must return `FALSE`.

Allocate a `ScrnInfoRec` for each active instance of the hardware found, and fill in the basic information, including the other driver entry points. This is best done with the `xf86ConfigIsaEntity()` helper function for ISA instances or `xf86ConfigPciEntity()` for PCI instances. These functions allocate a `ScrnInfoRec` for active entities. Optionally `xf86AllocateScreen()` function may also be used to allocate the `ScrnInfoRec`. Any of these functions take care of initialising fields to defined "unused" values.

Claim the entities for each instance of the hardware found. This prevents other drivers from claiming the same hardware.

Must leave hardware in the same state it found it in, and must not do any hardware initialisation.

All detection can be overridden via the config file, and that parsed information is available to the driver at this stage.

Returns `TRUE` if one or more instances are found, and `FALSE` otherwise.

```
int xf86MatchDevice(const char *drivername,

        GDevPtr **driversectlist)
```

This function takes the name of the driver and returns via `driversectlist` a list of device sections that match the driver name. The function return value is the number of matches found. If a fatal error is encountered the return value is `-1`.

The caller should use `xfree()` to free `*driversectlist` when it is no longer needed.

```
ScrnInfoPtr xf86AllocateScreen(DriverPtr drv, int flags)
```

This function allocates a new `ScrnInfoRec` in the `xf86Screens[]` array. This function is normally called by the video driver `ChipProbe()` functions. The return value is a pointer to the newly allocated `ScrnInfoRec`. The `scrnIndex`, `origIndex`, `module` and `drv` fields are initialised. The reference count in `drv` is incremented. The storage for any currently allocated "privates" pointers is also allocated and the `privates` field initialised (the privates data is of course not allocated or initialised). This function never returns on failure. If the allocation fails, the server exits with a fatal error. The flags value is not currently used, and should be set to zero.

At the completion of this, a list of `ScrnInfoRecs` have been allocated in the `xf86Screens[]` array, and the associated entities and fixed resources have been claimed. The following `ScrnInfoRec` fields must be initialised at this point:

```
driverVersion
driverName
scrnIndex(*)
origIndex(*)
drv(*)
module(*)
name
Probe
PreInit
ScreenInit
EnterVT
LeaveVT
numEntities
entityList
access
```

`(*)` These are initialised when the `ScrnInfoRec` is allocated, and not explicitly by the driver.

The following `ScrnInfoRec` fields must be initialised if the driver is going to use them:

```
SwitchMode
AdjustFrame
FreeScreen
ValidMode
```

5.9 Matching Screens

This is done at the start of the first server generation only.

After the Probe phase is finished, there will be some number of `ScrnInfoRecs`. These are then matched with the active **Screen** sections in the XF86Config, and those not having an active **Screen** section are deleted. If the number of remaining screens is 0, `InitOutput()` sets `screenInfo.numScreens` to `0` and returns.

At this point the following fields of the `ScrnInfoRecs` must be initialised:

```
confScreen
```

5.10 Allocate non-conflicting resources

This is done at the start of the first server generation only.

Before calling the drivers again, the resource information collected from the Probe phase is processed. This includes checking the extent of PCI resources for the probed devices, and resolving any conflicts in the relocatable PCI resources. It also reports conflicts, checks bus routing issues, and anything else that is needed to enable the entities for the next phase.

If any drivers registered an `EntityInit()` function during the Probe phase, then they are called here.

5.11 Sort the Screens and pre-check Monitor Information

This is done at the start of the first server generation only.

The list of screens is sorted to match the ordering requested in the config file.

The list of modes for each active monitor is checked against the monitor's parameters. Invalid modes are pruned.

5.12 PreInit

This is done at the start of the first server generation only.

For each `ScrnInfoRec`, enable access to the screens entities and call the `ChipPreInit()` function.

```
Bool ChipPreInit(ScrnInfoRec screen, int flags)
```

The purpose of this function is to find out all the information required to determine if the configuration is usable, and to initialise those parts of the `ScrnInfoRec` that can be set once at the beginning of the first server generation.

The number of entities registered for the screen should be checked against the expected number (most drivers expect only one). The entity information for each of them should be retrieved (with `xf86GetEntityInfo()`) and checked for the correct bus type and that none of the sharable resources registered during the Probe phase was rejected.

Access to resources for the entities that can be controlled in a device-independent way are enabled before this function is called. If the driver needs to access any resources that it has disabled in an `EntityInit()` function that it registered, then it may enable them here providing that it disables them before this function returns.

This includes probing for video memory, clocks, ramdac, and all other HW info that is needed. It includes determining the depth/bpp/visual and related info. It includes validating and determining the set of video modes that will be used (and anything that is required to determine that).

This information should be determined in the least intrusive way possible. The state of the HW must remain unchanged by this function. Although video memory (including MMIO) may be mapped within this function, it must be unmapped before returning. Driver specific information should be stored in a structure hooked into the `ScrnInfoRec`'s `driverPrivate` field. Any other modules which require persistent data (ie data that persists across server generations) should be initialised in this function, and they should allocate a "privates" index to hook their data into by calling `xf86AllocateScrnInfoPrivateIndex()`. The "privates" data is persistent.

Helper functions for some of these things are provided at the XFree86 common level, and the driver can choose to make use of them.

All additional resources that the screen needs must be registered here. This should be done with `xf86RegisterResources()`. If some of the fixed resources registered in the Probe phase are not needed or not decoded by the hardware when in the OPERATING server state, their status should be updated with `xf86SetOperatingState()`.

Modules may be loaded at any point in this function, and all modules that the driver will need must be loaded before the end of this function. Either the `xf86LoadSubModule()` or the `xf86LoadDrvSubModule()` function should be used to load modules depending on whether a `ScrnInfoRec` has been set up. A driver may unload a module within this function if it was only needed temporarily, and the `xf86UnloadSubModule()` function should be used to do that. Otherwise there is no need to explicitly unload modules because the loader takes care of module dependencies and will unload submodules automatically if/when the driver module is unloaded.

The bulk of the `ScrnInfoRec` fields should be filled out in this function.

`ChipPreInit()` returns `FALSE` when the configuration is unusable in some way (unsupported depth, no valid modes, not enough video memory, etc), and `TRUE` if it is usable.

It is expected that if the `ChipPreInit()` function returns `TRUE`, then the only reasons that subsequent stages in the driver might fail are lack or resources (like xalloc failures). All other possible reasons for failure should be determined by the `ChipPreInit()` function.

The `ScrnInfoRecs` for screens where the `ChipPreInit()` fails are removed. If none remain, `InitOutput()` sets `screenInfo.numScreens` to `0` and returns.

At this point, further fields of the `ScrnInfoRecs` would normally be filled in. Most are not strictly mandatory, but many are required by other layers and/or helper functions that the driver may choose to use. The documentation for those layers and helper functions indicates which they require.

The following fields of the `ScrnInfoRecs` should be filled in if the driver is going to use them:

```
monitor
display
depth
pixmapBPP
bitsPerPixel
weight                  (>8bpp only)
mask                    (>8bpp only)
offset                  (>8bpp only)
rgbBits                 (8bpp only)
gamma
defaultVisual
maxHValue
maxVValue
virtualX
virtualY
displayWidth
frameX0
frameY0
frameX1
frameY1
zoomLocked
modePool
modes
currentMode
progClock               (TRUE if clock is programmable)
chipset
ramdac
clockchip
numClocks               (if not programmable)
clock[]                 (if not programmable)
videoRam
biosBase
memBase
memClk
driverPrivate
chipID
chipRev
```

`pointer xf86LoadSubModule(ScrnInfoPtr pScrn, const char *name):` and `pointer xf86LoadDrvSubModule(DriverPtr drv, const char *name):`

> Load a module that a driver depends on. This function loads the module `name` as a sub module of the driver. The return value is a handle identifying the new module. If the load fails, the return value will be `NULL`. If a driver needs to explicitly unload a module it has loaded in this way, the return value must be saved and passed to `xf86UnloadSubModule()` when unloading.

`void xf86UnloadSubModule(pointer module)`

> Unloads the module referenced by `module`. `module` should be a pointer returned previously by `xf86LoadSubModule()` or `xf86LoadDrvSubModule()`.

5.13 Cleaning up Unused Drivers

At this point it is known which screens will be in use, and which drivers are being used. Unreferenced drivers (and modules they may have loaded) are unloaded here.

5.14 Consistency Checks

The parameters that must be global to the server, like pixmap formats, bitmap bit order, bitmap scanline unit and image byte order are compared for each of the screens. If a mismatch is found, the server exits with an appropriate message.

5.15 Check if Resource Control is Needed

Determine if resource access control is needed. This is the case if more than one screen is used. If necessary the RAC wrapper module is loaded.

5.16 AddScreen (ScreenInit)

At this point, the valid screens are known. `AddScreen()` is called for each of them, passing `ChipScreenInit()` as the argument. `AddScreen()` is a DIX function that allocates a new `screenInfo.screen[]` entry (aka `pScreen`), and does some basic initialisation of it. It then calls the `ChipScreenInit()` function, with `pScreen` as one of its arguments. If `ChipScreenInit()` returns `FALSE`, `AddScreen()` returns `-1`. Otherwise it returns the index of the screen. `AddScreen()` should only fail because of programming errors or failure to allocate resources (like memory). All configuration problems should be detected BEFORE this point.

```
Bool ChipScreenInit(int index, ScreenPtr pScreen,
          int argc, char **argv)
```

This is called at the start of each server generation.

Fill in all of `pScreen`, possibly doing some of this by calling ScreenInit functions from other layers like mi, framebuffers (cfb, etc), and extensions.

Decide which operations need to be placed under resource access control. The classes of operations are the frame buffer operations (`RAC_FB`), the pointer operations (`RAC_CURSOR`), the viewport change operations (`RAC_VIEWPORT`) and the colormap operations (`RAC_COLORMAP`). Any operation that requires resources which might be disabled during OPERATING state should be set to use RAC. This can be specified separately for memory and IO resources (the `racMemFlags` and `racIoFlags` fields of the `ScrnInfoRec` respectively).

Map any video memory or other memory regions.

Save the video card state. Enough state must be saved so that the original state can later be restored.

Initialise the initial video mode. The `ScrnInfoRec`'s `vtSema` field should be set to `TRUE` just prior to changing the video hardware's state.

The `ChipScreenInit()` function (or functions from other layers that it calls) should allocate entries in the `ScreenRec`'s `devPrivates` area by calling `AllocateScreenPrivateIndex()` if it needs per-generation storage. Since the `ScreenRec`'s `devPrivates` information is cleared for each server generation, this is the correct place to initialise it.

After `AddScreen()` has successfully returned, the following `ScrnInfoRec` fields are initialised:

```
pScreen
racMemFlags
racIoFlags
```

The `ChipScreenInit()` function should initialise the `CloseScreen` and `SaveScreen` fields of `pScreen`. The old value of `pScreen->CloseScreen` should be saved as part of the driver's per-screen private data, allowing it to be called from `ChipCloseScreen()`. This means that the existing `CloseScreen()` function is wrapped.

5.17 Finalising RAC Initialisation

After all the `ChipScreenInit()` functions have been called, each screen has registered its RAC requirements. This information is used to determine which shared resources are requested by more than one driver and set the access functions accordingly. This is done following these rules:

1. The sharable resources registered by each entity are compared. If a resource is registered by more than one entity the entity will be marked to indicate that it needs to share this resources type (IO or MEM).
2. A resource marked "disabled" during OPERATING state will be ignored entirely.
3. A resource marked "unused" will only conflict with an overlapping resource of an other entity if the second is actually in use during OPERATING state.
4. If an "unused" resource was found to conflict but the entity does not use any other resource of this type the entire resource type will be disabled for that entity.

5.18 Finishing InitOutput()

At this point `InitOutput()` is finished, and all the screens have been setup in their initial video mode.

5.19 Mode Switching

When a SwitchMode event is received, `ChipSwitchMode()` is called (when it exists):

```
Bool ChipSwitchMode(int index, DisplayModePtr mode, int flags)
```

Initialises the new mode for the screen identified by `index;`. The viewport may need to be adjusted also.

5.20 Changing Viewport

When a Change Viewport event is received, `ChipAdjustFrame()` is called (when it exists):

```
void ChipAdjustFrame(int index, int x, int y, int flags)
```

Changes the viewport for the screen identified by `index;`.

It should be noted that many chipsets impose restrictions on where the viewport may be placed in the virtual resolution, either for alignment reasons, or to prevent the start of the viewport from being positioned within a pixel (as can happen in a 24bpp mode). After calculating the value the chipset's panning registers need to be set to for non-DGA modes, this function should recalculate the ScrnInfoRec's `frameX0`, `frameY0`, `frameX1` and `frameY1` fields to correspond to that value. If this is not done, switching to another mode might cause the position of a hardware cursor to change.

5.21 VT Switching

When a VT switch event is received, `xf86VTSwitch()` is called. `xf86VTSwitch()` does the following:

On ENTER:

- enable port I/O access
- save and initialise the bus/resource state
- enter the SETUP server state
- calls `ChipEnterVT()` for each screen
- enter the OPERATING server state
- validate GCs
- Restore fb from saved pixmap for each screen
- Enable all input devices

On LEAVE:

- Save fb to pixmap for each screen
- validate GCs

- enter the SETUP server state
- calls `ChipLeaveVT()` for each screen
- disable all input devices
- restore bus/resource state
- disables port I/O access

`Bool ChipEnterVT(int index, int flags)`

This function should initialise the current video mode and initialise the viewport, turn on the HW cursor if appropriate, etc.

Should it re-save the video state before initialising the video mode?

`void ChipLeaveVT(int index, int flags)`

This function should restore the saved video state. If appropriate it should also turn off the HW cursor, and invalidate any pixmap/font caches.

Optionally, `ChipLeaveVT()` may also unmap memory regions. If so, `ChipEnterVT()` will need to remap them. Additionally, if an aperture used to access video memory is unmapped and remapped in this fashion, `ChipEnterVT()` will also need to notify the framebuffer layers of the aperture's new location in virtual memory. This is done with a call to the screen's `ModifyPixmapHeader()` function, as follows

```
(*pScreen->ModifyPixmapHeader)(pScrn->ppix,
        -1, -1, -1, -1, -1, NewApertureAddress);
```

where the ``ppix'' field in a ScrnInfoRec points to the pixmap used by the screen's `SaveRestoreImage()` function to hold the screen's contents while switched out.

Currently, aperture remapping, as described here, should not be attempted if the driver uses the `xf8_16bpp` or `xf8_32bpp` framebuffer layers. A pending restructuring of VT switching will address this restriction in the near future.

Other layers may wrap the `ChipEnterVT()` and `ChipLeaveVT()` functions if they need to take some action when these events are received.

5.22 End of server generation

At the end of each server generation, the DIX layer calls `ChipCloseScreen()` for each screen:

```
Bool ChipCloseScreen(int index, ScreenPtr pScreen)
```

This function should restore the saved video state and unmap the memory regions.

It should also free per-screen data structures allocated by the driver. Note that the persistent data held in the `ScrnInfoRec`'s `driverPrivate` field should not be freed here because it is needed by subsequent server generations.
The `ScrnInfoRec`'s `vtSema` field should be set to `FALSE` once the video HW state has been restored.

Before freeing the per-screen driver data the saved `CloseScreen` value should be restored to `pScreen->CloseScreen`, and that function should be called after freeing the data.

6. Optional Driver Functions

The functions outlined here can be called from the XFree86 common layer, but their presence is optional.

6.1 Mode Validation

When a mode validation helper supplied by the XFree86-common layer is being used, it can be useful to provide a function to check for hw specific mode constraints:

```
ModeStatus ChipValidMode(int index, DisplayModePtr mode,

        Bool verbose, int flags)
```

Check the passed mode for hw-specific constraints, and return the appropriate status value.

This function may also modify the effective timings and clock of the passed mode. These have been stored in the mode's `Crtc*` and `SynthClock` elements, and have already been adjusted for interlacing, doublescanning, multiscanning and clock multipliers and dividers. The function should not modify any other mode field, unless it wants to modify the mode timings reported to the user by `xf86PrintModes()`.

The function is called once for every mode in the XF86Config Monitor section assigned to the screen, with `flags` set to `MODECHECK_INITIAL`. It is subsequently called for every mode in the XF86Config Display subsection assigned to the screen, with `flags` set to `MODECHECK_FINAL`. In the second case, the mode will have successfully passed all other tests. In addition, the `ScrnInfoRec`'s `virtualX`, `virtualY` and `displayWidth` fields will have been set as if the mode to be validated were to be the last mode accepted.

In effect, calls with MODECHECK_INITIAL are intended for checks that do not depend on any mode other than the one being validated, while calls with MODECHECK_FINAL are intended for checks that may involve more than one mode.

6.2 Free screen data

When a screen is deleted prior to the completion of the ScreenInit phase the `ChipFreeScreen()` function is called when defined.

```
void ChipFreeScreen(int scrnindex, int flags)
```

Free any driver-allocated data that may have been allocated up to and including an unsuccessful `ChipScreenInit()` call. This would predominantly be data allocated by `ChipPreInit()` that persists across server generations. It would include the `driverPrivate`, and any "privates" entries that modules may have allocated.

7. Recommended driver functions

The functions outlined here are for internal use by the driver only. They are entirely optional, and are never accessed directly from higher layers. The sample function declarations shown here are just examples. The interface (if any) used is up to the driver.

7.1 Save

Save the video state. This could be called from `ChipScreenInit()` and (possibly) `ChipEnterVT()`.

```
void ChipSave(ScrnInfoPtr pScrn)
```

Saves the current state. This will only be saving pre-server states or states before returning to the server. There is only one current saved state per screen and it is stored in private storage in the screen.

7.2 Restore

Restore the original video state. This could be called from the `ChipLeaveVT()` and `ChipCloseScreen()` functions.

```
void ChipRestore(ScrnInfoPtr pScrn)
```

Restores the saved state from the private storage. Usually only used for restoring text modes.

7.3 Initialise Mode

Initialise a video mode. This could be called from the `ChipScreenInit()`, `ChipSwitchMode()` and `ChipEnterVT()` functions.

```
Bool ChipModeInit(ScrnInfoPtr pScrn, DisplayModePtr mode)
```

Programs the hardware for the given video mode.

8. Data and Data Structures

8.1 Command line data

Command line options are typically global, and are stored in global variables. These variables are read-only and are available to drivers via a function call interface. Most of these command line values are processed via helper functions to ensure that they are treated consistently by all drivers. The other means of access is provided for cases where the supplied helper functions might not be appropriate.

Some of them are:

```
xf86Verbose               verbosity level
xf86Bpp                   -bpp from the command line
xf86Depth                 -depth from the command line
xf86Weight                -weight from the command line
xf86Gamma                 -{r,g,b,}gamma from the command line
xf86FlipPixels            -flippixels from the command line
xf86ProbeOnly             -probeonly from the command line
defaultColorVisualClass   -cc from the command line
```

If we ever do allow for screen-specific command line options, we may need to rethink this.

These can be accessed in a read-only manner by drivers with the following functions:

`int xf86GetVerbosity()`

Returns the value of `xf86Verbose`.

`int xf86GetDepth()`

Returns the `-depth` command line setting. If not set on the command line, `-1` is returned.

`rgb xf86GetWeight()`

Returns the `-weight` command line setting. If not set on the command line, `{0, 0, 0}` is returned.

`Gamma xf86GetGamma()`

Returns the `-gamma` or `-rgamma`, `-ggamma`, `-bgamma` command line settings. If not set on the command line, `{0.0, 0.0, 0.0}` is returned.

`Bool xf86GetFlipPixels()`

Returns `TRUE` if `-flippixels` is present on the command line, and `FALSE` otherwise.

`const char *xf86GetServerName()`

Returns the name of the X server from the command line.

8.2 Data handling

Config file data contains parts that are global, and parts that are Screen specific. All of it is parsed into data structures that neither the drivers or most other parts of the server need to know about.

The global data is typically not required by drivers, and as such, most of it is stored in the private `xf86InfoRec`.

The screen-specific data collected from the config file is stored in screen, device, display, monitor-specific data structures that are separate from the `ScrnInfoRecs`, with the appropriate elements/fields hooked into the `ScrnInfoRecs` as required. The screen config data is held in `confScreenRec`, device data in the `GDevRec`, monitor data in the `MonRec`, and display data in the `DispRec`.

The XFree86 common layer's screen specific data (the actual data in use for each screen) is held in the `ScrnInfoRecs`. As has been outlined above, the `ScrnInfoRecs` are allocated at probe time, and it is the responsibility of the Drivers' `Probe()` and `PreInit()` functions to finish filling them in based on both data provided on the command line and data provided from the Config file. The precedence for this is:

command line -> config file -> probed/default data

For most things in this category there are helper functions that the drivers can use to ensure that the above precedence is consistently used.

As well as containing screen-specific data that the XFree86 common layer (including essential parts of the server infrastructure as well as helper functions) needs to access, it also contains some data that drivers use internally. When considering whether to add a new field to the `ScrnInfoRec`, consider the balance between the convenience of things that lots of drivers need and the size/obscurity of the `ScrnInfoRec`.

Per-screen driver specific data that cannot be accommodated with the static `ScrnInfoRec` fields is held in a driver-defined data structure, a pointer to which is assigned to the `ScrnInfoRec`'s `driverPrivate` field. This is per-screen data that persists across server generations (as does the bulk of the static `ScrnInfoRec` data). It would typically also include the video card's saved state.

Per-screen data for other modules that the driver uses (for example, the XAA module) that is reset for each server generation is hooked into the `ScrnInfoRec` through it's `privates` field.

Once it has stabilised, the data structures and variables accessible to video drivers will be documented here. In the meantime, those things defined in the `xf86.h` and `xf86str.h` files are visible to video drivers. Things defined in `xf86Priv.h` and `xf86Privstr.h` are NOT intended to be visible to video drivers, and it is an error for a driver to include those files.

8.3 Accessing global data

Some other global state information that the drivers may access via functions is as follows:

`Bool xf86ServerIsExiting()`

Returns `TRUE` if the server is at the end of a generation and is in the process of exiting, and `FALSE` otherwise.

`Bool xf86ServerIsResetting()`

Returns `TRUE` if the server is at the end of a generation and is in the process of resetting, and `FALSE` otherwise.

`Bool xf86ServerIsInitialising()`

Returns `TRUE` if the server is at the beginning of a generation and is in the process of initialising, and `FALSE` otherwise.

`Bool xf86ServerIsOnlyProbing()`

Returns `TRUE` if the -probeonly command line flag was specified, and `FALSE` otherwise.

`Bool xf86CaughtSignal()`

Returns `TRUE` if the server has caught a signal, and `FALSE` otherwise.

8.4 Allocating private data

A driver and any module it uses may allocate per-screen private storage in either the `ScreenRec` (DIX level) or `ScrnInfoRec` (XFree86 common layer level). `ScreenRec` storage persists only for a single server generation, and `ScrnInfoRec` storage persists across generations for the lifetime of the server.

The `ScreenRec` `devPrivates` data must be reallocated/initialised at the start of each new generation. This is normally done from the `ChipScreenInit()` function, and Init functions for other modules that it calls. Data allocated in this way should be freed by the driver's `ChipCloseScreen()` functions, and Close functions for other modules that it calls. A new `devPrivates` entry is allocated by calling the `AllocateScreenPrivateIndex()` function.

`int AllocateScreenPrivateIndex()`

> This function allocates a new element in the `devPrivates` field of all currently existing `ScreenRecs`. The return value is the index of this new element in the `devPrivates` array. The `devPrivates` field is of type `DevUnion`:
>
> ```
> typedef union _DevUnion {
>     pointer           ptr;
>     long              val;
>     unsigned long     uval;
>     pointer           (*fptr)(void);
> } DevUnion;
> ```
>
> which allows the element to be used for any of the above types. It is commonly used as a pointer to data that the caller allocates after the new index has been allocated.
>
> This function will return `-1` when there is an error allocating the new index.

The `ScrnInfoRec` `privates` data persists for the life of the server, so only needs to be allocated once. This should be done from the `ChipPreInit()` function, and Init functions for other modules that it calls. Data allocated in this way should be freed by the driver's `ChipFreeScreen()` functions, and Free functions for other modules that it calls. A new `privates` entry is allocated by calling the `xf86AllocateScrnInfoPrivateIndex()` function.

`int xf86AllocateScrnInfoPrivateIndex()`

> This function allocates a new element in the `privates` field of all currently existing `ScrnInfoRecs`. The return value is the index of this new element in the `privates` array. The `privates` field is of type `DevUnion`:
>
> ```
> typedef union _DevUnion {
>     pointer           ptr;
>     long              val;
>     unsigned long     uval;
>     pointer           (*fptr)(void);
> } DevUnion;
> ```
>
> which allows the element to be used for any of the above types. It is commonly used as a pointer to data that the caller allocates after the new index has been allocated.
>
> This function will not return when there is an error allocating the new index. When there is an error it will cause the server to exit with a fatal error. The similar function for allocation privates in the `ScreenRec` (`AllocateScreenPrivateIndex()`) differs in this respect by returning `-1` when the allocation fails.

9. Keeping Track of Bus Resources

9.1 Theory of Operation

The XFree86 common layer has knowledge of generic access control mechanisms for devices on certain bus systems (currently the PCI bus) as well as of methods to enable or disable access to the buses itself. Furthermore it can access information on resources decoded by these devices and if necessary modify it.

When first starting the Xserver collects all this information, saves it for restoration, checks it for consistency, and if necessary, corrects it. Finally it disables all resources on a generic level prior to calling any driver function.

When the `Probe()` function of each driver is called the device sections are matched against the devices found in the system. The driver may probe devices at this stage that cannot be identified by using device independent methods. Access to all resources that can be controlled in a device independent way is disabled. The `Probe()` function should register all non-relocatable resources at this stage. If a resource conflict is found between exclusive resources the driver will fail immediately. Optionally the driver might specify an `EntityInit()`, `EntityLeave()` and `EntityEnter()` function.

`EntityInit()` can be used to disable any shared resources that are not controlled by the generic access control functions. It is called prior to the PreInit phase regardless if an entity is active or not. When calling the `EntityInit()`, `EntityEnter()` and `EntityLeave()` functions the common level will disable access to all other entities on a generic level. Since the common level has no knowledge of device specific methods to disable access to resources it cannot be guaranteed that certain resources are not decoded by any other entity until the `EntityInit()` or `EntityEnter()` phase is finished. Device drivers should therefore register all those resources which they are going to disable. If these resources are never to be used by any driver function they may be flagged `ResInit` so that they can be removed from the resource list after processing all `EntityInit()` functions. `EntityEnter()` should disable decoding of all resources which are not registered as exclusive and which are not handled by the generic access control in the common level. The difference to `EntityInit()` is that the latter one is only called once during lifetime of the server. It can therefore be used to set up variables prior to disabling resources. `EntityLeave()` should restore the original state when exiting the server or switching to a different VT. It also needs to disable device specific access functions if they need to be disabled on server exit or VT switch. The default state is to enable them before giving up the VT.

In `PreInit()` phase each driver should check if any sharable resources it has registered during `Probe()` has been denied and take appropriate action which could simply be to fail. If it needs to access resources it has disabled during `EntitySetup()` it can do so provided it has registered these and will disable them before returning from `PreInit()`. This also applies to all other driver functions. Several functions are provided to request resource ranges, register these, correct PCI config space and add replacements for the generic access functions. Resources may be marked "disabled" or "unused" during OPERATING stage. Although these steps could also be performed in `ScreenInit()`, this is not desirable.

Following `PreInit()` phase the common level determines if resource access control is needed. This is the case if more than one screen is used. If necessary the RAC wrapper module is loaded. In `ScreenInit()` the drivers can decide which operations need to be placed under RAC. Available are the frame buffer operations, the pointer operations and the colormap operations. Any operation that requires resources which might be disabled during OPERATING state should be set to use RAC. This can be specified separately for memory and IO resources.

When `ScreenInit()` phase is done the common level will determine which shared resources are requested by more than one driver and set the access functions accordingly. This is done following these rules:

1. The sharable resources registered by each entity are compared. If a resource is registered by more than one entity the entity will be marked to need to share this resources type (`IO` or `MEM`).
2. A resource marked "disabled" during OPERATING state will be ignored entirely.
3. A resource marked "unused" will only conflicts with an overlapping resource of an other entity if the second is actually in use during OPERATING state.
4. If an "unused" resource was found to conflict however the entity does not use any other resource of this type the entire resource type will be disabled for that entity.

The driver has the choice among different ways to control access to certain resources:

1. It can rely on the generic access functions. This is probably the most common case. Here the driver only needs to register any resource it is going to use.
2. It can replace the generic access functions by driver specific ones. This will mostly be used in cases where no generic access functions are available. In this case the driver has to make sure these resources are disabled when entering the `PreInit()` stage. Since the replacement functions are registered in `PreInit()` the driver will have to enable these resources itself if it needs to access them during this state. The driver can specify if the replacement functions can control memory and/or I/O resources separately.
3. The driver can enable resources itself when it needs them. Each driver function enabling them needs to disable them before it will return. This should be used if a resource which can be controlled in a device dependent way is only required during SETUP state. This way it can be marked "unused" during OPERATING state.

A resource which is decoded during OPERATING state however never accessed by the driver should be marked unused.

Since access switching latencies are an issue during Xserver operation, the common level attempts to minimize the number of entities that need to be placed under RAC control. When a wrapped operation is called, the `EnableAccess()` function is called before control is passed on. `EnableAccess()` checks if a screen is under access control. If not it just establishes bus routing and returns. If the screen needs to be under access control, `EnableAccess()` determines which resource types (`MEM`, `IO`) are required. Then it tests if this access is already established. If so it simply returns. If not it disables the currently established access, fixes bus routing and enables access to all entities registered for this screen.

Whenever a mode switch or a VT-switch is performed the common level will return to SETUP state.

9.2 Resource Types

Resource have certain properties. When registering resources each range is accompanied by a flag consisting of the ORed flags of the different properties the resource has. Each resource range may be classified according to

- its physical properties i.e., if it addresses memory (`ResMem`) or I/O space (`ResIo`),
- if it addresses a block (`ResBlock`) or sparse (`ResSparse`) range,
- its access properties.

There are two known access properties:

- `ResExclusive` for resources which may not be shared with any other device and
- `ResShared` for resources which can be disabled and therefore can be shared.

If it is necessary to test a resource against any type a generic access type `ResAny` is provided. If this is set the resource will conflict with any resource of a different entity intersecting its range.

Further it can be specified that a resource is decoded however never used during any stage (`ResUnused`) or during OPERATING state (`ResUnusedOpr`). A resource only visible during the init functions (ie. `EntityInit()`, `EntityEnter()` and `EntityLeave()` should be registered with the flag `ResInit`. A resource that might conflict with background resource ranges may be flagged with `ResBios`. This might be useful when registering resources ranges that were assigned by the system Bios.

Several predefined resource lists are available for VGA and 8514/A resources in `common/xf86Resources.h`.

9.3 Available Functions

The functions provided for resource management are listed in their order of use in the driver.

9.3.1 Probe Phase

In this phase each driver detects those resources it is able to drive, creates an entity record for each of them, registers non-relocatable resources and allocates screens and adds the resources to screens.

Two helper functions are provided for matching device sections in the XF86Config file to the devices:

```
int xf86MatchPciInstances(const char *driverName, int vendorID,
              SymTabPtr chipsets, PciChipsets *PCIchipsets,
              GDevPtr *devList, int numDevs, DriverPtr drvp,
              int **foundEntities)
```

> This function finds matches between PCI cards that a driver supports and config file device sections. It is intended for use in the `ChipProbe()` function of drivers for PCI cards. Only probed PCI devices with a vendor ID matching `vendorID` are considered. `devList` and `numDevs` are typically those found from calling `xf86MatchDevice()`, and represent the active config file device sections relevant to the driver. `PCIchipsets` is a table that provides a mapping between the PCI device IDs, the driver's internal chipset tokens and a list of fixed resources.
>
> When a device section doesn't have a **BusID** entry it can only match the primary video device. Secondary devices are only matched with device sections that have a matching **BusID** entry.
>
> Once the preliminary matches have been found, a final match is confirmed by checking if the chipset override, ChipID override or probed PCI chipset type match one of those given in the `chipsets` and `PCIchipsets` lists. The `PCIchipsets` list includes a list of the PCI device IDs supported by the driver. The list should be terminated with an entry with PCI ID `-1`". The `chipsets` list is a table mapping the driver's internal chipset tokens to names, and should be terminated with a `NULL` entry. Only those entries with a corresponding entry in the `PCIchipsets` list are considered. The order of precedence is: config file chipset, config file ChipID, probed PCI device ID.
>
> In cases where a driver handles PCI chipsets with more than one vendor ID, it may set `vendorID` to `0`, and OR each devID in the list with (the vendor ID << 16).

Entity index numbers for confirmed matches are returned as an array via `foundEntities`. The PCI information, chipset token and device section for each match are found in the `EntityInfoRec` referenced by the indices.

The function return value is the number of confirmed matches. A return value of `-1` indicates an internal error. The returned `foundEntities` array should be freed by the driver with `xfree()` when it is no longer needed in cases where the return value is greater than zero.

```
int xf86MatchIsaInstances(const char *driverName,

        SymTabPtr chipsets, IsaChipsets *ISAchipsets,

        DriverPtr drvp, FindIsaDevProc FindIsaDevice,

        GDevPtr *devList, int numDevs, int **foundEntities)
```

This function finds matches between ISA cards that a driver supports and config file device sections. It is intended for use in the `ChipProbe()` function of drivers for ISA cards. `devList` and `numDevs` are typically those found from calling `xf86MatchDevice()`, and represent the active config file device sections relevant to the driver. `ISAchipsets` is a table that provides a mapping between the driver's internal chipset tokens and the resource classes. `FindIsaDevice` is a driver-provided function that probes the hardware and returns the chipset token corresponding to what was detected, and `-1` if nothing was detected.

If the config file device section contains a chipset entry, then it is checked against the `chipsets` list. When no chipset entry is present, the `FindIsaDevice` function is called instead.

Entity index numbers for confirmed matches are returned as an array via `foundEntities`. The chipset token and device section for each match are found in the `EntityInfoRec` referenced by the indices.

The function return value is the number of confirmed matches. A return value of `-1` indicates an internal error. The returned `foundEntities` array should be freed by the driver with `xfree()` when it is no longer needed in cases where the return value is greater than zero.

These two helper functions make use of several core functions that are available at the driver level:

```
Bool xf86ParsePciBusString(const char *busID, int *bus,
        int *device, int *func)
```

Takes a `BusID` string, and if it is in the correct format, returns the PCI `bus`, `device`, `func` values that it indicates. The format of the string is expected to be "PCI:bus:device:func" where each of 'bus', 'device' and 'func' are decimal integers. The ":func" part may be omitted, and the func value assumed to be zero, but this isn't encouraged. The "PCI" prefix may also be omitted. The prefix "AGP" is currently equivalent to the "PCI" prefix. If the string isn't a valid PCI BusID, the return value is `FALSE`.

```
Bool xf86ComparePciBusString(const char *busID, int bus,
        int device, int func)
```

Compares a `BusID` string with PCI `bus`, `device`, `func` values. If they match `TRUE` is returned, and `FALSE` if they don't.

```
Bool xf86ParseIsaBusString(const char *busID)
```

Compares a `BusID` string with the ISA bus ID string ("ISA" or "ISA:"). If they match `TRUE` is returned, and `FALSE` if they don't.

```
Bool xf86CheckPciSlot(int bus, int device, int func)
```

Checks if the PCI slot `bus:device:func` has been claimed. If so, it returns `FALSE`, and otherwise `TRUE`.

```
int xf86ClaimPciSlot(int bus, int device, int func, DriverPtr
drvp,
        int chipset, GDevPtr dev, Bool active)
```

This function is used to claim a PCI slot, allocate the associated entity record and initialise their data structures. The return value is the index of the newly allocated entity record, or `-1` if the claim fails. This function should always succeed if `xf86CheckPciSlot()` returned `TRUE` for the same PCI slot.

```
Bool xf86IsPrimaryPci(void)
```

This function returns `TRUE` if the primary card is a PCI device, and `FALSE` otherwise.

```
int xf86ClaimIsaSlot(DriverPtr drvp, int chipset,
        GDevPtr dev, Bool active)
```

This allocates an entity record entity and initialise the data structures. The return value is the index of the newly allocated entity record.

```
Bool xf86IsPrimaryIsa(void)
```

This function returns `TRUE` if the primary card is an ISA (non-PCI) device, and `FALSE` otherwise.

Two helper functions are provided to aid configuring entities:

```
ScrnInfoPtr xf86ConfigPciEntity(ScrnInfoPtr pScrn,
          int scrnFlag, int entityIndex,
          PciChipsets *p_chip,
          resList res, EntityProc init,
          EntityProc enter, EntityProc leave,
          pointer private)
ScrnInfoPtr xf86ConfigIsaEntity(ScrnInfoPtr pScrn,
          int scrnFlag, int entityIndex,
          IsaChipsets *i_chip,
          resList res, EntityProc init,
          EntityProc enter, EntityProc leave,
          pointer private)
```

These functions are used to register the non-relocatable resources for an entity, and the optional entity-specific `Init`, `Enter` and `Leave` functions. Usually the list of fixed resources is obtained from the Isa/PciChipsets lists. However an additional list of resources may be passed. Generally this is not required. For active entities a `ScrnInfoRec` is allocated if the `pScrn` argument is `NULL`. The return value is `TRUE` when successful. The init, enter, leave functions are defined as follows:

```
typedef void (*EntityProc)(int entityIndex,
          pointer private)
```

They are passed the entity index and a pointer to a private scratch area. This can be set up during `Probe()` and its address can be passed to `xf86ConfigIsaEntity()` and `xf86ConfigPciEntity()` as the last argument.

These two helper functions make use of several core functions that are available at the driver level:

`void xf86ClaimFixedResources(resList list, int entityIndex)`

> This function registers the non-relocatable resources which cannot be disabled and which therefore would cause the server to fail immediately if they were found to conflict. It also records non-relocatable but sharable resources for processing after the `Probe()` phase.

`Bool xf86SetEntityFuncs(int entityIndex, EntityProc init,`

`EntityProc enter, EntityProc leave, pointer)`

> This function registers with an entity the `init`, `enter`, `leave` functions along with the pointer to their private area.

`void xf86AddEntityToScreen(ScrnInfoPtr pScrn, int entityIndex)`

> This function associates the entity referenced by `entityIndex` with the screen.

9.3.2 PreInit Phase

During this phase the remaining resources should be registered. `PreInit()` should call `xf86GetEntityInfo()` to obtain a pointer to an `EntityInfoRec` for each entity it is able to drive and check if any resource are listed in its `resources` field. If resources registered in the Probe phase have been rejected in the post-Probe phase (`resources` is non-`NULL`), then the driver should decide if it can continue without using these or if it should fail.

`EntityInfoPtr xf86GetEntityInfo(int entityIndex)`

> This function returns a pointer to the `EntityInfoRec` referenced by `entityIndex`. The returned `EntityInfoRec` should be freed with `xfree()` when no longer needed.

Several functions are provided to simplify resource registration:

`Bool xf86IsEntityPrimary(int entityIndex)`

> This function returns `TRUE` if the entity referenced by `entityIndex` is the primary display device (i.e., the one initialised at boot time and used in text mode).

`Bool xf86IsScreenPrimary(int scrnIndex)`

> This function returns `TRUE` if the primary entity is registered with the screen referenced by `scrnIndex`.

`pciVideoPtr xf86GetPciInfoForEntity(int entityIndex)`

> This function returns a pointer to the `pciVideoRec` for the specified entity. If the entity is not a PCI device, `NULL` is returned.

The primary function for registration of resources is:

```
resPtr xf86RegisterResources(int entityIndex, resList list,
        int access)
```

This function tries to register the resources in `list`. If list is `NULL` it tries to determine the resources automatically. This only works for entities that provide a generic way to read out the resource ranges they decode. So far this is only the case for PCI devices. By default the PCI resources are registered as shared (`ResShared`) if the driver wants to set a different access type it can do so by specifying the access flags in the third argument. A value of `0` means to use the default settings. If for any reason the resource broker is not able to register some of the requested resources the function will return a pointer to a list of the failed ones. In this case the driver may be able to move the resource to different locations. In case of PCI bus entities this is done by passing the list of failed resources to `xf86ReallocatePciResources()`. When the registration succeeds, the return value is `NULL`.

```
resPtr xf86ReallocatePciResources(int entityIndex, resPtr pRes)
```

This function takes a list of PCI resources that need to be reallocated and returns `NULL` when all relocations are successful. `xf86RegisterResources()` should be called again to register the relocated resources with the broker. If the reallocation fails, a list of the resources that could not be relocated is returned.

Two functions are provided to obtain a resource range of a given type:

```
resRange xf86GetBlock(long type, memType size,
        memType window_start, memType window_end,
        memType align_mask, resPtr avoid)
```

This function tries to find a block range of size `size` and type `type` in a window bound by `window_start` and `window_end` with the alignment specified in `align_mask`. Optionally a list of resource ranges which should be avoided within the window can be supplied. On failure a zero-length range of type `ResEnd` will be returned.

```
resRange xf86GetSparse(long type, memType fixed_bits,
        memType decode_mask, memType address_mask,
        resPtr avoid)
```

This function is like the previous one, but attempts to find a sparse range instead of a block range. Here three values have to be specified: the `address_mask` which marks all bits of the mask part of the address, the `decode_mask` which masks out the bits which are hardcoded and are therefore not available for relocation and the values of the fixed bits. The function tries to find a base that satisfies the given condition. If the function fails it will return a zero range of type `ResEnd`. Optionally it might be passed a list of resource ranges to avoid.

Some PCI devices are broken in the sense that they return invalid size information for a certain resource. In this case the driver can supply the correct size and make sure that the resource range allocated for the card is large enough to hold the address range decoded by the card. The function `xf86FixPciResource()` can be used to do this:

```
Bool xf86FixPciResource(int entityIndex, unsigned int prt,

                    CARD32 alignment, long type)
```

> This function fixes a PCI resource allocation. The `prt` parameter contains the number of the PCI base register that needs to be fixed (`0-5`, and `6` for the BIOS base register). The size is specified by the alignment. Since PCI resources need to span an integral range of size `2^n`, the alignment also specifies the number of addresses that will be decoded. If the driver specifies a type mask it can override the default type for PCI resources which is `ResShared`. The resource broker needs to know that to find a matching resource range. This function should be called before calling `xf86RegisterResources()`. The return value is `TRUE` when the function succeeds.

```
Bool xf86CheckPciMemBase(pciVideoPtr pPci, memType base)
```

> This function checks that the memory base address specified matches one of the PCI base address register values for the given PCI device. This is mostly used to check that an externally provided base address (e.g., from a config file) matches an actual value allocated to a device.

The driver may replace the generic access control functions for an entity. This is done with the `xf86SetAccessFuncs()`:

```
void xf86SetAccessFuncs(EntityInfoPtr pEnt,

        xf86SetAccessFuncPtr funcs,

        xf86SetAccessFuncPtr oldFuncs)
```

with:

```
typedef struct {
    xf86AccessPtr mem;
    xf86AccessPtr io;
    xf86AccessPtr io_mem;
} xf86SetAccessFuncRec, *xf86SetAccessFuncPtr;
```

The driver can pass three functions: one for I/O access, one for memory access and one for combined memory and I/O access. If the memory access and combined access functions are identical the common level assumes that the memory access cannot be controlled independently of I/O access, if the I/O access function and the combined access functions are the same it is assumed that I/O can not be controlled independently. If memory and I/O have to be controlled together all three values should be the same. If a non `NULL` value is passed as third argument it is interpreted as an address where to store the old access record. If the third argument is `NULL` it will be assumed that the generic access should be enabled before replacing the access functions. Otherwise it will be disabled. The driver may enable them itself using the returned values. It should do this from its replacement access functions as the generic access may be disabled by the common level on certain occasions. If replacement functions are specified they must control all resources of the specific type registered for the entity.

To find out if a specific resource range conflicts with another resource the `xf86ChkConflict()` function may be used:

```
memType xf86ChkConflict(resRange *rgp, int entityIndex)
```

This function checks if the resource range `rgp` of for the specified entity conflicts with with another resource. If a conflict is found, the address of the start of the conflict is returned. The return value is zero when there is no conflict.

The OPERATING state properties of previously registered fixed resources can be set with the `xf86SetOperatingState()` function:

```
resPtr xf86SetOperatingState(resList list, int entityIndex,
        int mask)
```

This function is used to set the status of a resource during OPERATING state. `list` holds a list to which `mask` is to be applied. The parameter `mask` may have the value `ResUnusedOpr` and `ResDisableOpr`. The first one should be used if a resource isn't used by the driver during OPERATING state although it is decoded by the device, while the latter one indicates that the resource is not decoded during OPERATING state. Note that the resource ranges have to match those specified during registration. If a range has been specified starting at `A` and ending at `B` and suppose `C` us a value satisfying `A < C < B` one may not specify the resource range `(A,B)` by splitting it into two ranges `(A,C)` and `(C,B)`.

The following two functions are provided for special cases:

```
void xf86RemoveEntityFromScreen(ScrnInfoPtr pScrn, int entityIndex)
```

This function may be used to remove an entity from a screen. This only makes sense if a screen has more than one entity assigned or the screen is to be deleted. No test is made if the screen has any entities left.

```
void xf86DeallocateResourcesForEntity(int entityIndex, long type)
```

This function deallocates all resources of a given type registered for a certain entity from the resource broker list.

9.3.3 ScreenInit Phase

All that is required in this phase is to setup the RAC flags. Note that it is also permissible to set these flags up in the PreInit phase. The RAC flags are held in the `racIoFlags` and `racMemFlags` fields of the `ScrnInfoRec` for each screen. They specify which graphics operations might require the use of shared resources. This can be specified separately for memory and I/O resources. The available flags are defined in `rac/xf86RAC.h`. They are:

`RAC_FB`

for framebuffer operations (including hw acceleration)

`RAC_CURSOR`

for Cursor operations (??? I'm not sure if we need this for SW cursor it depends on which level the sw cursor is drawn)

`RAC_COLORMAP`

for colormap operations

`RAC_VIEWPORT`

for the call to `ChipAdjustFrame()`

The flags are ORed together.

10. Config file "Option" entries

Option entries are permitted in most sections and subsections of the config file. There are two forms of option entries:

Option "option-name"
A boolean option.

Option "option-name" "option-value"
An option with an arbitrary value.

The option entries are handled by the parser, and a list of the parsed options is included with each of the appropriate data structures that the drivers have access to. The data structures used to hold the option information are opaque to the driver, and a driver must not access the option data directly. Instead, the common layer provides a set of functions that may be used to access, check and manipulate the option data.

First, the low level option handling functions. In most cases drivers would not need to use these directly.

```
pointer xf86FindOption(pointer options, const char *name)
```

Takes a list of options and an option name, and returns a handle for the first option entry in the list matching the name. Returns `NULL` if no match is found.

```
char *xf86FindOptionValue(pointer options, const char *name)
```

Takes a list of options and an option name, and returns the value associated with the first option entry in the list matching the name. If the matching option has no value, an empty string (`""`) is returned. Returns `NULL` if no match is found.

```
void xf86MarkOptionUsed(pointer option)
```

Takes a handle for an option, and marks that option as used.

```
void xf86MarkOptionUsedByName(pointer options, const char *name)
```

Takes a list of options and an option name and marks the first option entry in the list matching the name as used.

Next, the higher level functions that most drivers would use.

`void xf86CollectOptions(ScrnInfoPtr pScrn, pointer extraOpts)`

Collect the options from each of the config file sections used by the screen (`pScrn`) and return the merged list as `pScrn->options`. This function requires that `pScrn->confScreen`, `pScrn->display`, `pScrn->monitor`, `pScrn->numEntities`, and `pScrn->entityList` are initialised. `extraOpts` may optionally be set to an additional list of options to be combined with the others. The order of precedence for options is `extraOpts`, display, confScreen, monitor, device.

```
void xf86ProcessOptions(int scrnIndex, pointer options,

        OptionInfoPtr optinfo)
```

Processes a list of options according to the information in the array of `OptionInfoRecs` (`optinfo`). The resulting information is stored in the `value` fields of the appropriate `optinfo` entries. The `found` fields are set to `TRUE` when an option with a value of the correct type if found, and `FALSE` otherwise. The `type` field is used to determine the expected value type for each option. Each option in the list of options for which there is a name match (but not necessarily a value type match) is marked as used. Warning messages are printed when option values don't match the types specified in the optinfo data.

NOTE: If this function is called before a driver's screen number is known (e.g., from the `ChipProbe()` function) a `scrnIndex` value of `-1` should be used.

NOTE 2: Given that this function stores into the `OptionInfoRecs` pointed to by `optinfo`, the caller should ensure the `OptionInfoRecs` are (re-)initialised before the call, especially if the caller expects to use the predefined option values as defaults.

The `OptionInfoRec` is defined as follows:

```
typedef struct {
    double freq;
    int units;
} OptFrequency;

typedef union {
    unsigned long       num;
    char *              str;
    double              realnum;
    Bool                bool;
    OptFrequency        freq;
} ValueUnion;

typedef enum {
    OPTV_NONE = 0,
    OPTV_INTEGER,
    OPTV_STRING,  /* a non-empty string */
    OPTV_ANYSTR,  /* Any string, including an empty one */
    OPTV_REAL,
    OPTV_BOOLEAN,
    OPTV_FREQ
} OptionValueType;

typedef enum {
    OPTUNITS_HZ = 1,
    OPTUNITS_KHZ,
```

```
        OPTUNITS_MHZ
    } OptFreqUnits;

    typedef struct {
        int                 token;
        const char*         name;
        OptionValueType     type;
        ValueUnion          value;
        Bool                found;
    } OptionInfoRec, *OptionInfoPtr;
```

`OPTV_FREQ` can be used for options values that are frequencies. These values are a floating point number with an optional unit name appended. The unit name can be one of "Hz", "kHz", "k", "MHz", "M". The multiplier associated with the unit is stored in `freq.units`, and the scaled frequency is stored in `freq.freq`. When no unit is specified, `freq.units` is set to `0`, and `freq.freq` is unscaled.

Typical usage is to setup an array of `OptionInfoRecs` with all fields initialised. The `value` and `found` fields get set by `xf86ProcessOptions()`. For cases where the value parsing is more complex, the driver should specify `OPTV_STRING`, and parse the string itself. An example of using this option handling is included in the *Sample Driver* (section 20., page 87) section.

`void xf86ShowUnusedOptions(int scrnIndex, pointer options)`

Prints out warning messages for each option in the list of options that isn't marked as used. This is intended to show options that the driver hasn't recognised. It would normally be called near the end of the `ChipScreenInit()` function, but only when `serverGeneration == 1`.

```
OptionInfoPtr xf86TokenToOptinfo(const OptionInfoRec *table,

        int token)
```

Returns a pointer to the `OptionInfoRec` in `table` with a token field matching `token`. Returns `NULL` if no match is found.

`Bool xf86IsOptionSet(const OptionInfoRec *table, int token)`

Returns the `found` field of the `OptionInfoRec` in `table` with a `token` field matching `token`. This can be used for options of all types. Note that for options of type `OPTV_BOOLEAN`, it isn't sufficient to check this to determine the value of the option. Returns `FALSE` if no match is found.

`char *xf86GetOptValString(const OptionInfoRec *table, int token)`

Returns the `value.str` field of the `OptionInfoRec` in `table` with a token field matching `token`. Returns `NULL` if no match is found.

```
Bool xf86GetOptValInteger(const OptionInfoRec *table, int token,

        int *value)
```

Returns via `*value` the `value.num` field of the `OptionInfoRec` in `table` with a `token` field matching `token`. `*value` is only changed when a match is found so it can be safely initialised with a default prior to calling this function. The function return value is as for `xf86IsOptionSet()`.

```
Bool xf86GetOptValULong(const OptionInfoRec *table, int token,

        unsigned long *value)
```

Like `xf86GetOptValInteger()`, except the value is treated as an `unsigned long`.

```
Bool xf86GetOptValReal(const OptionInfoRec *table, int token,

        double *value)
```

Like `xf86GetOptValInteger()`, except that `value.realnum` is used.

```
Bool xf86GetOptValFreq(const OptionInfoRec *table, int token,

        OptFreqUnits expectedUnits, double *value)
```

Like `xf86GetOptValInteger()`, except that the `value.freq` data is returned. The frequency value is scaled to the units indicated by `expectedUnits`. The scaling is exact when the units were specified explicitly in the option's value. Otherwise, the `expectedUnits` field is used as a hint when doing the scaling. In this case, values larger than `1000` are assumed to have be specified in the next smallest units. For example, if the Option value is "10000" and expectedUnits is `OPTUNITS_MHZ`, the value returned is `10`.

```
Bool xf86GetOptValBool(const OptionInfoRec *table, int token,
Bool *value)
```

This function is used to check boolean options (`OPTV_BOOLEAN`). If the function return value is `FALSE`, it means the option wasn't set. Otherwise `*value` is set to the boolean value indicated by the option's value. No option `value` is interpreted as `TRUE`. Option values meaning `TRUE` are "1", "yes", "on", "true", and option values meaning `FALSE` are "0", "no", "off", "false". Option names both with the "no" prefix in their names, and with that prefix removed are also checked and handled in the obvious way. `*value` is not changed when the option isn't present. It should normally be set to a default value before calling this function.

```
Bool xf86ReturnOptValBool(const OptionInfoRec *table, int
token, Bool def)
```

This function is used to check boolean options (`OPTV_BOOLEAN`). If the option is set, its value is returned. If the options is not set, the default value specified by `def` is returned. The option interpretation is the same as for `xf86GetOptValBool()`.

```
int xf86NameCmp(const char *s1, const char *s2)
```

This function should be used when comparing strings from the config file with expected values. It works like `strcmp()`, but is not case sensitive and space, tab, and '_' characters are ignored in the comparison. The use of this function isn't restricted to parsing option values. It may be used anywhere where this functionality required.

11. Modules, Drivers, Include Files and Interface Issues

NOTE: this section is incomplete.

11.1 Include files

The following include files are typically required by video drivers:

All drivers should include these:

```
"xf86.h"

"xf86_OSproc.h"

"xf86_ansic.h"

"xf86Resources.h"
```

Wherever inb/outb (and related things) are used the following should be included:

```
"compiler.h"
```

Note: in drivers, this must be included after `"xf86_ansic.h"`.

Drivers that need to access PCI vendor/device definitions need this:

```
"xf86PciInfo.h"
```

Drivers that need to access the PCI config space need this:

```
"xf86Pci.h"
```

Drivers that initialise a SW cursor need this:

```
"mipointer.h"
```

All drivers implementing backing store need this:

```
"mibstore.h"
```

All drivers using the mi colourmap code need this:

```
"micmap.h"
```

If a driver uses the vgahw module, it needs this:

```
"vgaHW.h"
```

Drivers supporting VGA or Hercules monochrome screens need:

```
"xf1bpp.h"
```

Drivers supporting VGA or EGC 16-colour screens need:

```
"xf4bpp.h"
```

Drivers using cfb need:

```
#define PSZ 8

#include "cfb.h"

#undef PSZ
```

Drivers supporting bpp 16, 24 or 32 with cfb need one or more of:

`"cfb16.h"`

`"cfb24.h"`

`"cfb32.h"`

If a driver uses XAA, it needs these:

`"xaa.h"`

`"xaalocal.h"`

If a driver uses the fb manager, it needs this:

`"xf86fbman.h"`

Non-driver modules should include `"xf86_ansic.h"` to get the correct wrapping of ANSI C/libc functions.

All modules must NOT include any system include files, or the following:

`"xf86Priv.h"`

`"xf86Privstr.h"`

`"xf86_OSlib.h"`

`"Xos.h"`

In addition, "xf86_libc.h" must not be included explicitly. It is included implicitly by "xf86_ansic.h".

12. Offscreen Memory Manager

Management of offscreen video memory may be handled by the XFree86 framebuffer manager. Once the offscreen memory manager is running, drivers or extensions may allocate, free or resize areas of offscreen video memory using the following functions (definitions taken from `xf86fbman.h`):

```
typedef struct _FBArea {
    ScreenPtr    pScreen;
    BoxRec       box;
    int          granularity;
    void         (*MoveAreaCallback)(struct _FBArea*, struct _FBArea*)
    void         (*RemoveAreaCallback)(struct _FBArea*)
    DevUnion     devPrivate;
} FBArea, *FBAreaPtr;

typedef void (*MoveAreaCallbackProcPtr)(FBAreaPtr from, FBAreaPtr to)
typedef void (*RemoveAreaCallbackProcPtr)(FBAreaPtr)

FBAreaPtr xf86AllocateOffscreenArea (
    ScreenPtr pScreen,
    int width, int height,
    int granularity,
    MoveAreaCallbackProcPtr MoveAreaCallback,
    RemoveAreaCallbackProcPtr RemoveAreaCallback,
    pointer privData
)

void xf86FreeOffscreenArea (FBAreaPtr area)

Bool xf86ResizeOffscreenArea (
   FBAreaPtr area
   int w, int h
)
```

The function:

```
Bool xf86FBManagerRunning(ScreenPtr pScreen)
```

can be used by an extension to check if the driver has initialized the memory manager. The manager is not available if this returns `FALSE` and the functions above will all fail.

`xf86AllocateOffscreenArea()` can be used to request a rectangle of dimensions `width` x `height` (in pixels) from unused offscreen memory. `granularity` specifies that the leftmost edge of the rectangle must lie on some multiple of `granularity` pixels. A granularity of zero means the same thing as a granularity of one - no alignment preference. A `MoveAreaCallback` can be provided to notify the requester when the offscreen area is moved. If no `MoveAreaCallback` is supplied then the area is considered to be immovable. The `privData` field will be stored in the manager's internal structure for that allocated area and will be returned to the requester in the `FBArea` passed via the `MoveAreaCallback`. An optional `RemoveAreaCallback` is provided. If the driver provides this it indicates that the area should be allocated with a lower priority. Such an area may be removed when a higher priority request (one that doesn't have a `RemoveAreaCallback`) is made. When this function is called, the driver will have an opportunity to do whatever cleanup it needs to do to deal with the loss of the area, but it must finish its cleanup before the function exits since the offscreen memory manager will free the area immediately after.

`xf86AllocateOffscreenArea()` returns `NULL` if it was unable to allocate the requested area. When no longer needed, areas should be freed with `xf86FreeOffscreenArea()`.

`xf86ResizeOffscreenArea()` resizes an existing `FBArea`. `xf86ResizeOffscreenArea()` returns `TRUE` if the resize was successful. If `xf86ResizeOffscreenArea()` returns `FALSE`, the original `FBArea` is left unmodified. Resizing an area maintains the area's original `granularity`, `devPrivate`, and `MoveAreaCallback`. `xf86ResizeOffscreenArea()` has considerably less overhead than freeing the old area then reallocating the new size, so it should be used whenever possible.

The function:

```
Bool xf86QueryLargestOffscreenArea(
          ScreenPtr pScreen,
          int *width, int *height,
          int granularity,
          int preferences,
          int priority
)
```

is provided to query the width and height of the largest single `FBArea` allocatable given a particular priority. `preferences` can be one of the following to indicate whether width, height or area should be considered when determining which is the largest single `FBArea` available.

```
FAVOR_AREA_THEN_WIDTH
FAVOR_AREA_THEN_HEIGHT
FAVOR_WIDTH_THEN_AREA
FAVOR_HEIGHT_THEN_AREA
```

`priority` is one of the following:

`PRIORITY_LOW`

> Return the largest block available without stealing anyone else's space. This corresponds to the priority of allocating a `FBArea` when a `RemoveAreaCallback` is provided.

`PRIORITY_NORMAL`

> Return the largest block available if it is acceptable to steal a lower priority area from someone. This corresponds to the priority of allocating a `FBArea` without providing a `RemoveAreaCallback`.

`PRIORITY_EXTREME`

> Return the largest block available if all `FBAreas` that aren't locked down were expunged from memory first. This corresponds to any allocation made directly after a call to `xf86PurgeUnlockedOffscreenAreas()`.

The function:

```
Bool xf86PurgeUnlockedOffscreenAreas(ScreenPtr pScreen)
```

is provided as an extreme method to free up offscreen memory. This will remove all removable `FBArea` allocations.

Initialization of the XFree86 framebuffer manager is done via

```
Bool xf86InitFBManager(ScreenPtr pScreen, BoxPtr FullBox)
```

`FullBox` represents the area of the framebuffer that the manager is allowed to manage. This is typically a box with a width of `pScrn->displayWidth` and a height of as many lines as can be fit within the total video memory, however, the driver can reserve areas at the extremities by passing a smaller area to the manager.

`xf86InitFBManager()` must be called before XAA is initialized since XAA uses the manager for it's pixmap cache.

An alternative function is provided to allow the driver to initialize the framebuffer manager with a Region rather than a box.

```
Bool xf86InitFBManagerRegion(ScreenPtr pScreen,

          RegionPtr FullRegion)
```

`xf86InitFBManagerRegion()`, unlike `xf86InitFBManager()`, does not remove the area used for the visible screen so that area should not be included in the region passed to the function. `xf86InitFBManagerRegion()` is useful when non-contiguous areas are available to be managed, and is required when multiple framebuffers are stored in video memory (as in the case where an overlay of a different depth is stored as a second framebuffer in offscreen memory).

13. Colormap Handling

A generic colormap handling layer is provided within the XFree86 common layer. This layer takes care of most of the details, and only requires a function from the driver that loads the hardware palette when required. To use the colormap layer, a driver calls the `xf86HandleColormaps()` function.

```
Bool xf86HandleColormaps(ScreenPtr pScreen, int maxColors,

          int sigRGBbits, LoadPaletteFuncPtr loadPalette,

          SetOverscanFuncPtr setOverscan, unsigned int flags)
```

This function must be called after the default colormap has been initialised. The `pScrn->gamma` field must also be initialised, preferably by calling `xf86SetGamma()`. `maxColors` is the number of entries in the palette. `sigRGBbits` is the size in bits of each color component in the DAC's palette. `loadPalette` is a driver-provided function for loading a colormap into the hardware, and is described below. `setOverscan` is an optional function that may be provided when the overscan color is an index from the standard LUT and when it needs to be adjusted to keep it as close to black as possible. The `setOverscan` function programs the overscan index. It shouldn't normally be used for depths other than 8. `setOverscan` should be set to `NULL` when it isn't needed. `flags` may be set to the following (which may be ORed together):

`CMAP_PALETTED_TRUECOLOR`

the TrueColor visual is paletted and is just a special case of DirectColor. This flag is only valid for `bpp > 8`.

`CMAP_RELOAD_ON_MODE_SWITCH`

reload the colormap automatically after mode switches. This is useful for when the driver is resetting the hardware during mode switches and corrupting or erasing the hardware palette.

`CMAP_LOAD_EVEN_IF_OFFSCREEN`

reload the colormap even if the screen is switched out of the server's VC. The palette is *not* reloaded when the screen is switched back in, nor after mode switches. This is useful when the driver needs to keep track of palette changes.

The colormap layer normally reloads the palette after VT enters so it is not necessary for the driver to save and restore the palette when switching VTs. The driver must, however, still save the initial palette during server start up and restore it during server exit.

```
void LoadPalette(ScrnInfoPtr pScrn, int numColors, int *indices,
          LOCO *colors, VisualPtr pVisual)
```

`LoadPalette()` is a driver-provided function for loading a colormap into hardware. `colors` is the array of RGB values that represent the full colormap. `indices` is a list of index values into the colors array. These indices indicate the entries that need to be updated. `numColors` is the number of the indices to be updated.

```
void SetOverscan(ScrnInfoPtr pScrn, int overscan)
```

`SetOverscan()` is a driver-provided function for programming the `overscan` index. As described above, it is normally only appropriate for LUT modes where all colormap entries are available for the display, but where one of them is also used for the overscan (typically 8bpp for VGA compatible LUTs). It isn't required in cases where the overscan area is never visible.

14. DPMS Extension

Support code for the DPMS extension is included in the XFree86 common layer. This code provides an interface between the main extension code, and a means for drivers to initialise DPMS when they support it. One function is available to drivers to do this initialisation, and it is always available, even when the DPMS extension is not supported by the core server (in which case it returns a failure result).

```
Bool xf86DPMSInit(ScreenPtr pScreen, DPMSSetProcPtr set, int flags)
```

This function registers a driver's DPMS level programming function `set`. It also checks `pScrn->options` for the "dpms" option, and when present marks DPMS as being enabled for that screen. The `set` function is called whenever the DPMS level changes, and is used to program the requested level. `flags` is currently not used, and should be `0`. If the initialisation fails for any reason, including when there is no DPMS support in the core server, the function returns `FALSE`.

Drivers that implement DPMS support must provide the following function, that gets called when the DPMS level is changed:

`void ChipDPMSSet(ScrnInfoPtr pScrn, int level, int flags)`

Program the DPMS level specified by `level`. Valid values of `level` are `DPMSModeOn`, `DPMSModeStandby`, `DPMSModeSuspend`, `DPMSModeOff`. These values are defined in `"extensions/dpms.h"`.

15. DGA Extension

Drivers can support the XFree86 Direct Graphics Architecture (DGA) by filling out a structure of function pointers and a list of modes and passing them to DGAInit.

`Bool DGAInit(ScreenPtr pScreen, DGAFunctionPtr funcs,`

`DGAModePtr modes, int num)`

```
/** The DGAModeRec **/

typedef struct {
  int num;
  DisplayModePtr mode;
  int flags;
  int imageWidth;
  int imageHeight;
  int pixmapWidth;
  int pixmapHeight;
  int bytesPerScanline;
  int byteOrder;
  int depth;
  int bitsPerPixel;
  unsigned long red_mask;
  unsigned long green_mask;
  unsigned long blue_mask;
  int viewportWidth;
  int viewportHeight;
  int xViewportStep;
  int yViewportStep;
  int maxViewportX;
  int maxViewportY;
  int viewportFlags;
  int offset;
  unsigned char *address;
  int reserved1;
  int reserved2;
} DGAModeRec, *DGAModePtr;
```

`num`

Can be ignored. The DGA DDX will assign these numbers.

`mode`

A pointer to the `DisplayModeRec` for this mode.

`flags`

The following flags are defined and may be OR'd together:

`DGA_CONCURRENT_ACCESS`

Indicates that the driver supports concurrent graphics accelerator and linear framebuffer access.

`DGA_FILL_RECT`

`DGA_BLIT_RECT`

`DGA_BLIT_RECT_TRANS`

Indicates that the driver supports the FillRect, BlitRect or BlitTransRect functions in this mode.

`DGA_PIXMAP_AVAILABLE`

Indicates that Xlib may be used on the framebuffer. This flag will usually be set unless the driver wishes to prohibit this for some reason.

`DGA_INTERLACED`

`DGA_DOUBLESCAN`

Indicates that these are interlaced or double scan modes.

`imageWidth`

`imageHeight`

These are the dimensions of the linear framebuffer accessible by the client.

`pixmapWidth`

`pixmapHeight`

These are the dimensions of the area of the framebuffer accessible by the graphics accelerator.

`bytesPerScanline`

Pitch of the framebuffer in bytes.

`byteOrder`

Usually the same as `pScrn->imageByteOrder`.

`depth`

The depth of the framebuffer in this mode.

`bitsPerPixel`

The number of bits per pixel in this mode.

`red_mask`

`green_mask`

`blue_mask`

The RGB masks for this mode, if applicable.

`viewportWidth`

`viewportHeight`

Dimensions of the visible part of the framebuffer. Usually `mode->HDisplay` and `mode->VDisplay`.

`xViewportStep`

`yViewportStep`

The granularity of x and y viewport positions that the driver supports in this mode.

`maxViewportX`

`maxViewportY`

The maximum viewport position supported by the driver in this mode.

`viewportFlags`

The following may be OR'd together:

`DGA_FLIP_IMMEDIATE`

The driver supports immediate viewport changes.

`DGA_FLIP_RETRACE`

The driver supports viewport changes at retrace.

`offset`

The offset into the linear framebuffer that corresponds to pixel (0,0) for this mode.

`address`

The virtual address of the framebuffer as mapped by the driver. This is needed when DGA_PIXMAP_AVAILABLE is set.

```
/** The DGAFunctionRec **/

typedef struct {
  Bool (*OpenFramebuffer)(
       ScrnInfoPtr pScrn,
       char **name,
       unsigned char **mem,
       int *size,
       int *offset,
       int *extra
  );
  void (*CloseFramebuffer)(ScrnInfoPtr pScrn);
  Bool (*SetMode)(ScrnInfoPtr pScrn, DGAModePtr pMode);
  void (*SetViewport)(ScrnInfoPtr pScrn, int x, int y, int flags);
  int  (*GetViewport)(ScrnInfoPtr pScrn);
```

```
    void (*Sync)(ScrnInfoPtr);
    void (*FillRect)(
         ScrnInfoPtr pScrn,
         int x, int y, int w, int h,
         unsigned long color
    );
    void (*BlitRect)(
         ScrnInfoPtr pScrn,
         int srcx, int srcy,
         int w, int h,
         int dstx, int dsty
    );
    void (*BlitTransRect)(
         ScrnInfoPtr pScrn,
         int srcx, int srcy,
         int w, int h,
         int dstx, int dsty,
         unsigned long color
    );
  } DGAFunctionRec, *DGAFunctionPtr;
```

`Bool OpenFramebuffer (pScrn, name, mem, size, offset, extra)`

`OpenFramebuffer()` should pass the client everything it needs to know to be able to open the framebuffer. These parameters are OS specific and their meanings are to be interpreted by an OS specific client library.

`name`

The name of the device to open or `NULL` if there is no special device to open. A `NULL` name tells the client that it should open whatever device one would usually open to access physical memory.

`mem`

The physical address of the start of the framebuffer.

`size`

The size of the framebuffer in bytes.

`offset`

Any offset into the device, if applicable.

`flags`

Any additional information that the client may need. Currently, only the `DGA_NEED_ROOT` flag is defined.

`void CloseFramebuffer (pScrn)`

`CloseFramebuffer()` merely informs the driver (if it even cares) that client no longer needs to access the framebuffer directly. This function is optional.

`Bool SetMode (pScrn, pMode)`

`SetMode()` tells the driver to initialize the mode passed to it. If `pMode` is `NULL`, then the driver should restore the original pre-DGA mode.

`void SetViewport (pScrn, x, y, flags)`

`SetViewport()` tells the driver to make the upper left-hand corner of the visible screen correspond to coordinate `(x,y)` on the framebuffer. `Flags` currently defined are:

`DGA_FLIP_IMMEDIATE`

The viewport change should occur immediately.

`DGA_FLIP_RETRACE`

The viewport change should occur at the vertical retrace, but this function should return sooner if possible.

The `(x,y)` locations will be passed as the client specified them, however, the driver is expected to round these locations down to the next supported location as specified by the `xViewportStep` and `yViewportStep` for the current mode.

`int GetViewport (pScrn)`

`GetViewport()` gets the current page flip status. Set bits in the returned int correspond to viewport change requests still pending. For instance, set bit zero if the last SetViewport request is still pending, bit one if the one before that is still pending, etc.

`void Sync (pScrn)`

This function should ensure that any graphics accelerator operations have finished. This function should not return until the graphics accelerator is idle.

`void FillRect (pScrn, x, y, w, h, color)`

This optional function should fill a rectangle `w × h` located at `(x,y)` in the given color.

`void BlitRect (pScrn, srcx, srcy, w, h, dstx, dsty)`

This optional function should copy an area `w × h` located at `(srcx,srcy)` to location `(dstx,dsty)`. This function will need to handle copy directions as appropriate.

`void BlitTransRect (pScrn, srcx, srcy, w, h, dstx, dsty, color)`

This optional function is the same as BlitRect except that pixels in the source corresponding to the color key `color` should be skipped.

16. The XFree86 X Video Extension (Xv) Device Dependent Layer

XFree86 offers the X Video Extension which allows clients to treat video as any another primitive and "Put" video into drawables. By default, the extension reports no video adaptors as being available since the DDX layer has not been initialized. The driver can initialize the DDX layer by filling out one or more `XF86VideoAdaptorRecs` as described later in this document and passing a list of `XF86VideoAdaptorPtr` pointers to the following function:

```
Bool xf86XVScreenInit(

        ScreenPtr pScreen,

        XF86VideoAdaptorPtr *adaptPtrs,

        int num)
```

After doing this, the extension will report video adaptors as being available, providing the data in their respective `XF86VideoAdaptorRecs` was valid. `xf86XVScreenInit()` *copies* data from the structure passed to it so the driver may free it after the initialization. At the moment, the DDX only supports rendering into Window drawables. Pixmap rendering will be supported after a sufficient survey of suitable hardware is completed.

The `XF86VideoAdaptorRec`:

```
typedef struct {
        unsigned int type;
        int flags;
        char *name;
        int nEncodings;
        XF86VideoEncodingPtr pEncodings;
        int nFormats;
        XF86VideoFormatPtr pFormats;
        int nPorts;
        DevUnion *pPortPrivates;
        int nAttributes;
        XF86AttributePtr pAttributes;
        int nImages;
        XF86ImagePtr pImages;
        PutVideoFuncPtr PutVideo;
        PutStillFuncPtr PutStill;
        GetVideoFuncPtr GetVideo;
        GetStillFuncPtr GetStill;
        StopVideoFuncPtr StopVideo;
        SetPortAttributeFuncPtr SetPortAttribute;
        GetPortAttributeFuncPtr GetPortAttribute;
        QueryBestSizeFuncPtr QueryBestSize;
        PutImageFuncPtr PutImage;
        QueryImageAttributesFuncPtr QueryImageAttributes;
} XF86VideoAdaptorRec, *XF86VideoAdaptorPtr;
```

Each adaptor will have its own XF86VideoAdaptorRec. The fields are as follows:

`type`

This can be any of the following flags OR'd together.

`XvInputMask` `XvOutputMask`

These refer to the target drawable and are similar to a Window's class. `XvInputMask` indicates that the adaptor can put video into a drawable. `XvOutputMask` indicates that the adaptor can get video from a drawable.

`XvVideoMask` `XvStillMask` `XvImageMask`

These indicate that the adaptor supports video, still or image primitives respectively.

`XvWindowMask` `XvPixmapMask`

These indicate the types of drawables the adaptor is capable of rendering into. At the moment, Pixmap rendering is not supported and the `XvPixmapMask` flag is ignored.

`flags`

Currently, the following flags are defined:

`VIDEO_NO_CLIPPING`

This indicates that the video adaptor does not support clipping. The driver will never receive "Put" requests where less than the entire area determined by `drw_x`, `drw_y`, `drw_w` and `drw_h` is visible. This flag does not apply to "Get" requests. Hardware that is incapable of clipping "Gets" may punt or get the extents of the clipping region passed to it.

`VIDEO_INVERT_CLIPLIST`

This indicates that the video driver requires the clip list to contain the regions which are obscured rather than the regions which are are visible.

`VIDEO_OVERLAID_STILLS`

Implementing PutStill for hardware that does video as an overlay can be awkward since it's unclear how long to leave the video up for. When this flag is set, StopVideo will be called whenever the destination gets clipped or moved so that the still can be left up until then.

`VIDEO_OVERLAID_IMAGES`

Same as `VIDEO_OVERLAID_STILLS` but for images.

`VIDEO_CLIP_TO_VIEWPORT`

Indicates that the clip region passed to the driver functions should be clipped to the visible portion of the screen in the case where the viewport is smaller than the virtual desktop.

`name`

The name of the adaptor.

`nEncodings`

`pEncodings`

The number of encodings the adaptor is capable of and pointer to the `XF86VideoEncodingRec` array. The `XF86VideoEncodingRec` is described later on. For drivers that only support XvImages there should be an encoding named "XV_IMAGE" and the width and height should specify the maximum size source image supported.

`nFormats`

`pFormats`

The number of formats the adaptor is capable of and pointer to the `XF86VideoFormatRec` array. The `XF86VideoFormatRec` is described later on.

`nPorts`

`pPortPrivates`

The number of ports is the number of separate data streams which the adaptor can handle simultaneously. If you have more than one port, the adaptor is expected to be able to render into more than one window at a time. `pPortPrivates` is an array of pointers or ints - one for each port. A port's private data will be passed to the driver any time the port is requested to do something like put the video or stop the video. In the case where there may be many ports, this enables the driver to know which port the request is intended for. Most commonly, this will contain a pointer to the data structure containing information about the port. In Xv, all ports on a particular adaptor are expected to be identical in their functionality.

`nAttributes`

`pAttributes`

The number of attributes recognized by the adaptor and a pointer to the array of `XF86AttributeRecs`. The `XF86AttributeRec` is described later on.

`nImages`

`pImages`

The number of `XF86ImageRecs` supported by the adaptor and a pointer to the array of `XF86ImageRecs`. The `XF86ImageRec` is described later on.

`PutVideo PutStill GetVideo GetStill StopVideo SetPortAttribute GetPortAttribute QueryBestSize PutImage QueryImageAttributes`

These functions define the DDX->driver interface. In each case, the pointer `data` is passed to the driver. This is the port private for that port as described above. All fields are required except under the following conditions:

1. `PutVideo`, `PutStill` and the image routines `PutImage` and `QueryImageAttributes` are not required when the adaptor type does not contain `XvInputMask`.

2. `GetVideo` and `GetStill` are not required when the adaptor type does not contain `XvOutputMask`.
3. `GetVideo` and `PutVideo` are not required when the adaptor type does not contain `XvVideoMask`.
4. `GetStill` and `PutStill` are not required when the adaptor type does not contain `XvStillMask`.
5. `PutImage` and `QueryImageAttributes` are not required when the adaptor type does not contain `XvImageMask`.

With the exception of `QueryImageAttributes`, these functions should return `Success` if the operation was completed successfully. They can return `XvBadAlloc` otherwise. `QueryImageAttributes` returns the size of the XvImage queried.

If the `VIDEO_NO_CLIPPING` flag is set, the `clipBoxes` may be ignored by the driver. `ClipBoxes` is an `X-Y` banded region identical to those used throughout the server. The clipBoxes represent the visible portions of the area determined by `drw_x`, `drw_y`, `drw_w` and `drw_h` in the Get/Put function. The boxes are in screen coordinates, are guaranteed not to overlap and an empty region will never be passed. If the driver has specified `VIDEO_INVERT_CLIPLIST`, `clipBoxes` will indicate the areas of the primitive which are obscured rather than the areas visible.

```
typedef  int (* PutVideoFuncPtr)( ScrnInfoPtr pScrn,

          short vid_x, short vid_y, short drw_x, short drw_y,

          short vid_w, short vid_h, short drw_w, short drw_h,

          RegionPtr clipBoxes, pointer data )
```

This indicates that the driver should take a subsection `vid_w` by `vid_h` at location `(vid_x,vid_y)` from the video stream and direct it into the rectangle `drw_w` by `drw_h` at location `(drw_x,drw_y)` on the screen, scaling as necessary. Due to the large variations in capabilities of the various hardware expected to be used with this extension, it is not expected that all hardware will be able to do this exactly as described. In that case the driver should just do ‘‘the best it can,’’ scaling as closely to the target rectangle as it can without rendering outside of it. In the worst case, the driver can opt to just not turn on the video.

```
typedef  int (* PutStillFuncPtr)( ScrnInfoPtr pScrn,

          short vid_x, short vid_y, short drw_x, short drw_y,

          short vid_w, short vid_h, short drw_w, short drw_h,

          RegionPtr clipBoxes, pointer data )
```

This is same as `PutVideo` except that the driver should place only one frame from the stream on the screen.

```
typedef int (* GetVideoFuncPtr)( ScrnInfoPtr pScrn,

          short vid_x, short vid_y, short drw_x, short drw_y,
```

```
        short vid_w, short vid_h, short drw_w, short drw_h,
        RegionPtr clipBoxes, pointer data )
```

This is same as `PutVideo` except that the driver gets video from the screen and outputs it. The driver should do the best it can to get the requested dimensions correct without reading from an area larger than requested.

```
typedef int (* GetStillFuncPtr)( ScrnInfoPtr pScrn,
        short vid_x, short vid_y, short drw_x, short drw_y,
        short vid_w, short vid_h, short drw_w, short drw_h,
        RegionPtr clipBoxes, pointer data )
```

This is the same as `GetVideo` except that the driver should place only one frame from the screen into the output stream.

```
typedef void (* StopVideoFuncPtr)(ScrnInfoPtr pScrn,
        pointer data, Bool cleanup)
```

This indicates the driver should stop displaying the video. This is used to stop both input and output video. The `cleanup` field indicates that the video is being stopped because the client requested it to stop or because the server is exiting the current VT. In that case the driver should deallocate any offscreen memory areas (if there are any) being used to put the video to the screen. If `cleanup` is not set, the video is being stopped temporarily due to clipping or moving of the window, etc... and video will likely be restarted soon so the driver should not deallocate any offscreen areas associated with that port.

```
typedef int (* SetPortAttributeFuncPtr)(ScrnInfoPtr pScrn,
        Atom attribute,INT32 value, pointer data)
typedef int (* GetPortAttributeFuncPtr)(ScrnInfoPtr pScrn,
        Atom attribute,INT32 *value, pointer data)
```

A port may have particular attributes such as hue, saturation, brightness or contrast. Xv clients set and get these attribute values by sending attribute strings (Atoms) to the server. Such requests end up at these driver functions. It is recommended that the driver provide at least the following attributes mentioned in the Xv client library docs:

```
XV_ENCODING
XV_HUE
XV_SATURATION
XV_BRIGHTNESS
XV_CONTRAST
```

but the driver may recognize as many atoms as it wishes. If a requested attribute is unknown by the driver it should return `BadMatch`. `XV_ENCODING` is the attribute intended to let the client specify which video encoding the particular port should be using (see the description of `XF86VideoEncodingRec` below). If the requested encoding is unsupported, the driver should return `XvBadEncoding`. If the value lies outside the advertised range `BadValue` may be returned. `Success` should be returned otherwise.

```
typedef void (* QueryBestSizeFuncPtr)(ScrnInfoPtr pScrn,

          Bool motion, short vid_w, short vid_h,

          short drw_w, short drw_h,

          unsigned int *p_w, unsigned int *p_h, pointer data)
```

`QueryBestSize` provides the client with a way to query what the destination dimensions would end up being if they were to request that an area `vid_w` by `vid_h` from the video stream be scaled to rectangle of `drw_w` by `drw_h` on the screen. Since it is not expected that all hardware will be able to get the target dimensions exactly, it is important that the driver provide this function.

```
typedef  int (* PutImageFuncPtr)( ScrnInfoPtr pScrn,

          short src_x, short src_y, short drw_x, short drw_y,

          short src_w, short src_h, short drw_w, short drw_h,

          int image, char *buf, short width, short height,

          Bool sync, RegionPtr clipBoxes, pointer data )
```

This is similar to `PutStill` except that the source of the video is not a port but the data stored in a system memory buffer at `buf`. The data is in the format indicated by the `image` descriptor and represents a source of size `width` by `height`. If `sync` is TRUE the driver should not return from this function until it is through reading the data from `buf`. Returning when `sync` is TRUE indicates that it is safe for the data at `buf` to be replaced, freed, or modified.

```
typedef   int   (*   QueryImageAttributesFuncPtr)(   ScrnInfoPtr
pScrn,

          int image, short *width, short *height,

          int *pitches, int *offsets)
```

This function is called to let the driver specify how data for a particular `image` of size `width` by `height` should be stored. Sometimes only the size and corrected width and height are needed. In that case `pitches` and `offsets` are NULL. The size of the memory required for the image is returned by this function. The `width` and `height` of the requested image can be altered by the driver to reflect format limitations (such as component sampling periods that are larger than one). If `pitches` and `offsets` are not NULL, these will be arrays with as many elements in them as there are planes in the `image` format. The driver should specify the pitch (in bytes) of each scanline in the particular plane as well as the offset to that plane (in bytes) from the beginning of the image.

The XF86VideoEncodingRec:

```
typedef struct {
        int id;
        char *name;
        unsigned short width, height;
        XvRationalRec rate;
} XF86VideoEncodingRec, *XF86VideoEncodingPtr;
```

The `XF86VideoEncodingRec` specifies what encodings the adaptor can support. Most of this data is just informational and for the client's benefit, and is what will be reported by `XvQueryEncodings`. The `id` field is expected to be a unique identifier to allow the client to request a certain encoding via the `XV_ENCODING` attribute string.

The XF86VideoFormatRec:

```
typedef struct {
        char  depth;
        short class;
} XF86VideoFormatRec, *XF86VideoFormatPtr;
```

This specifies what visuals the video is viewable in. `depth` is the depth of the visual (not bpp). `class` is the visual class such as `TrueColor`, `DirectColor` or `PseudoColor`. Initialization of an adaptor will fail if none of the visuals on that screen are supported.

The XF86AttributeRec:

```
typedef struct {
        int   flags;
        int   min_value;
        int   max_value;
        char  *name;
} XF86AttributeListRec, *XF86AttributeListPtr;
```

Each adaptor may have an array of these advertising the attributes for its ports. Currently defined flags are `XvGettable` and `XvSettable` which may be OR'd together indicating that attribute is ''gettable'' or ''settable'' by the client. The `min` and `max` field specify the valid range for the value. `Name` is a text string describing the attribute by name.

The XF86ImageRec:

```
typedef struct {
        int id;
        int type;
        int byte_order;
        char guid[16];
        int bits_per_pixel;
        int format;
        int num_planes;

        /* for RGB formats */
        int depth;
        unsigned int red_mask;
        unsigned int green_mask;
        unsigned int blue_mask;

        /* for YUV formats */
        unsigned int y_sample_bits;
        unsigned int u_sample_bits;
        unsigned int v_sample_bits;
        unsigned int horz_y_period;
        unsigned int horz_u_period;
        unsigned int horz_v_period;
        unsigned int vert_y_period;
        unsigned int vert_u_period;
        unsigned int vert_v_period;
        char component_order[32];
        int scanline_order;
} XF86ImageRec, *XF86ImagePtr;
```

XF86ImageRec describes how video source data is laid out in memory. The fields are as follows:

`id`

> This is a unique descriptor for the format. It is often good to set this value to the FOURCC for the format when applicable.

`type`

> This is `XvRGB` or `XvYUV`.

`byte_order`

> This is `LSBFirst` or `MSBFirst`.

`guid`

> This is the Globally Unique IDentifier for the format. When not applicable, all characters should be NULL.

`bits_per_pixel`

> The number of bits taken up (but not necessarily used) by each pixel. Note that for some planar formats which have fractional bits per pixel (such as IF09) this number may be rounded _down_.

`format`

> This is `XvPlanar` or `XvPacked`.

`num_planes`

> The number of planes in planar formats. This should be set to one for

packed formats.

`depth`

The significant bits per pixel in RGB formats (analgous to the depth of a pixmap format).

`red_mask green_mask blue_mask`

The red, green and blue bitmasks for packed RGB formats.

`y_sample_bits u_sample_bits v_sample_bits`

The y, u and v sample sizes (in bits).

`horz_y_period horz_u_period horz_v_period`

The y, u and v sampling periods in the horizontal direction.

`vert_y_period vert_u_period vert_v_period`

The y, u and v sampling periods in the vertical direction.

`component_order`

Uppercase ascii characters representing the order that samples are stored within packed formats. For planar formats this represents the ordering of the planes. Unused characters in the 32 byte string should be set to NULL.

`scanline_order`

This is `XvTopToBottom` or `XvBottomToTop`.

Since some formats (particular some planar YUV formats) may not be completely defined by the parameters above, the guid, when available, should provide the most accurate description of the format.

17. The Loader

This section describes the interfaces to the module loader. The loader interfaces can be divided into two groups: those that are only available to the XFree86 common layer, and those that are also available to modules.

17.1 Loader Overview

The loader is capable of loading modules in a range of object formats, and knowledge of these formats is built in to the loader. Knowledge of new object formats can be added to the loader in a straightforward manner. This makes it possible to provide OS-independent modules (for a given CPU architecture type). In addition to this, the loader can load modules via the OS-provided `dlopen(3)` service where available. Such modules are not platform independent, and the semantics of `dlopen()` on most systems results in significant limitations in the use of modules of this type. Support for `dlopen()` modules in the loader is primarily for experimental and development purposes.

Symbols exported by the loader (on behalf of the core X server) to modules are determined at compile time. Only those symbols explicitly exported are available to modules. All external symbols of loaded modules are exported to other modules, and to the core X server. The loader can be requested to check for unresolved symbols at any time, and the action to be taken for unresolved symbols can be controlled by the caller of the loader. Typically the caller identifies which symbols can safely remain unresolved and which cannot.

NOTE: Now that ISO-C allows pointers to functions and pointers to data to have different internal representations, some of the following interfaces will need to be revisited.

17.2 Semi-private Loader Interface

The following is the semi-private loader interface that is available to the XFree86 common layer.

```
void LoaderInit(void)
```

The `LoaderInit()` function initialises the loader, and it must be called once before calling any other loader functions. This function initialises the tables of exported symbols, and anything else that might need to be initialised.

```
void LoaderSetPath(const char *path)
```

The `LoaderSetPath()` function initialises a default module search path. This must be called if calls to other functions are to be made without explicitly specifying a module search path. The search path `path` must be a string of one or more comma separated absolute paths. Modules are expected to be located below these paths, possibly in subdirectories of these paths.

```
pointer LoadModule(const char *module, const char *path,
          const char **subdirlist, const char **patternlist,
          pointer options, const XF86ModReqInfo * modreq,
          int *errmaj, int *errmin)
```

The `LoadModule()` function loads the module called `module`. The return value is a module handle, and may be used in future calls to the loader that require a reference to a loaded module. The module name `module` is normally the module's canonical name, which doesn't contain any directory path information, or any object/library file prefixes of suffixes. Currently a full pathname and/or filename is also accepted. This might change. The other parameters are:

`path`

An optional comma-separated list of module search paths. When `NULL`, the default search path is used.

`subdirlist`

An optional `NULL` terminated list of subdirectories to search. When `NULL`, the default built-in list is used (refer to `stdSubdirs` in `loadmod.c`). The default list is also substituted for entries in `subdirlist` with the value `DEFAULT_LIST`. This makes is possible to augment the default list instead of replacing it. Subdir elements must be relative, and must not contain `".."`. If any violate this requirement, the load fails.

`patternlist`

An optional `NULL` terminated list of POSIX regular expressions used to connect module filenames with canonical module names. Each regex should contain exactly one subexpression that corresponds to the canonical module name. When `NULL`, the default built-in list is used (refer to `stdPatterns` in `loadmod.c`). The default list is also substituted for entries in `patternlist` with the value `DEFAULT_LIST`. This makes it possible to augment the default list instead of replacing it.

`options`

An optional parameter that is passed to the newly loaded module's `SetupProc` function (if it has one). This argument is normally a `NULL` terminated list of `Options`, and must be interpreted that way by modules loaded directly by the XFree86 common layer. However, it may be used for application-specific parameter passing in other situations.

When loading "external" modules (modules that don't have the standard entry point, for example a special shared library) the options parameter can be set to `EXTERN_MODULE` to tell the loader not to reject the module when it doesn't find the standard entry point.

`modreq`

An optional `XF86ModReqInfo*` containing version/ABI/vendor information to requirements to check the newly loaded module against. The main purpose of this is to allow the loader to verify that a module of the correct type/version before running its `SetupProc` function.

The `XF86ModReqInfo` struct is defined as follows:

```
typedef struct {
        CARD8        majorversion;  /* MAJOR_UNSPEC */
        CARD8        minorversion;  /* MINOR_UNSPEC */
        CARD16       patchlevel;    /* PATCH_UNSPEC */
        const char * abiclass;      /* ABI_CLASS_NONE */
        CARD32       abiversion;    /* ABI_VERS_UNSPEC */
        const char * moduleclass;   /* MOD_CLASS_NONE */
} XF86ModReqInfo;
```

The information here is compared against the equivalent information in the module's `XF86ModuleVersionInfo` record (which is described below). The values in comments above indicate "don't care" settings for each of the fields. The comparisons made are as follows:

`majorversion`

Must match the module's majorversion exactly.

`minorversion`

The module's minor version must be no less than this value. This comparison is only made if `majorversion` is specified and matches.

`patchlevel`

The module's patchlevel must be no less than this value. This comparison is only made if `minorversion` is specified and matches.

`abiclass`

String must match the module's abiclass string.

`abiversion`

Must be consistent with the module's abiversion (major equal, minor no older).

`moduleclass`

String must match the module's moduleclass string.

`errmaj`

An optional pointer to a variable holding the major part or the error code. When provided, `*errmaj` is filled in when `LoadModule()` fails.

`errmin`

Like `errmaj`, but for the minor part of the error code.

`void UnloadModule(pointer mod)`

This function unloads the module referred to by the handle mod. All child modules are also unloaded recursively. This function must not be used to directly unload modules that are child modules (i.e., those that have been loaded with the `LoadSubModule()` described below).

17.3 Module Requirements

Modules must provide information about themselves to the loader, and may optionally provide entry points for "setup" and "teardown" functions (those two functions are referred to here as `SetupProc` and `TearDownProc`).

The module information is contained in the `XF86ModuleVersionInfo` struct, which is defined as follows:

```
typedef struct {
    const char * modname;      /* name of module, e.g. "foo" */
    const char * vendor;       /* vendor specific string */
    CARD32       _modinfo1_;   /* constant MODINFOSTRING1/2 to find */
    CARD32       _modinfo2_;   /* infoarea with a binary editor/sign tool */
    CARD32       xf86version;  /* contains XF86_VERSION_CURRENT */
    CARD8        majorversion; /* module-specific major version */
    CARD8        minorversion; /* module-specific minor version */
    CARD16       patchlevel;   /* module-specific patch level */
    const char * abiclass;     /* ABI class that the module uses */
    CARD32       abiversion;   /* ABI version */
    const char * moduleclass;  /* module class */
    CARD32       checksum[4];  /* contains a digital signature of the */
                               /* version info structure */
} XF86ModuleVersionInfo;
```

The fields are used as follows:

`modname`

The module's name. This field is currently only for informational purposes, but the loader may be modified in future to require it to match the module's canonical name.

`vendor`

The module vendor. This field is for informational purposes only.

`_modinfo1_`

This field holds the first part of a signature that can be used to locate this structure in the binary. It should always be initialised to `MODINFOSTRING1`.

`_modinfo2_`

This field holds the second part of a signature that can be used to locate this structure in the binary. It should always be initialised to `MODINFOSTRING2`.

`xf86version`

The XFree86 version against which the module was compiled. This is mostly for informational/diagnostic purposes. It should be initialised to `XF86_VERSION_CURRENT`, which is defined in `xf86Version.h`.

`majorversion`

The module-specific major version. For modules where this version is used for more than simply informational purposes, the major version should only change (be incremented) when ABI incompatibilities are introduced, or ABI components are removed.

`minorversion`

The module-specific minor version. For modules where this version is used for more than simply informational purposes, the minor version should only change (be incremented) when ABI additions are made in a backward compatible way. It should be reset to zero when the major version is increased.

`patchlevel`

The module-specific patch level. The patch level should increase with new revisions of the module where there are no ABI changes, and it should be reset to zero when the minor version is increased.

`abiclass`

The ABI class that the module requires. The class is specified as a string for easy extensibility. It should indicate which (if any) of the X server's built-in ABI classes that the module relies on, or a third-party ABI if appropriate. Built-in ABI classes currently defined are:

`ABI_CLASS_NONE`

no class

`ABI_CLASS_ANSIC`

only requires the ANSI C interfaces

`ABI_CLASS_VIDEODRV`

requires the video driver ABI

`ABI_CLASS_XINPUT`

requires the XInput driver ABI

`ABI_CLASS_EXTENSION`

requires the extension module ABI

`ABI_CLASS_FONT`

requires the font module ABI

`abiversion`

The version of abiclass that the module requires. The version consists of major and minor components. The major version must match and the minor version must be no newer than that provided by the server or parent module. Version identifiers for the built-in classes currently defined are:

`ABI_ANSIC_VERSION`

`ABI_VIDEODRV_VERSION`

`ABI_XINPUT_VERSION`

`ABI_EXTENSION_VERSION`

`ABI_FONT_VERSION`

`moduleclass`

This is similar to the abiclass field, except that it defines the type of module rather than the ABI it requires. For example, although all video drivers require the video driver ABI, not all modules that require the video driver ABI are video drivers. This distinction can be made with the moduleclass. Currently pre-defined module classes are:

`MOD_CLASS_NONE`

`MOD_CLASS_VIDEODRV`

`MOD_CLASS_XINPUT`

`MOD_CLASS_FONT`

`MOD_CLASS_EXTENSION`

`checksum`

Not currently used.

The module version information, and the optional `SetupProc` and `TearDownProc` entry points are found by the loader by locating a data object in the module called "modnameModuleData", where "modname" is the canonical name of the module. Modules must contain such a data object, and it must be declared with global scope, be compile-time initialised, and is of the following type:

```
typedef struct {
    XF86ModuleVersionInfo *     vers;
    ModuleSetupProc             setup;
    ModuleTearDownProc          teardown;
} XF86ModuleData;
```

The vers parameter must be initialised to a pointer to a correctly initialised `XF86ModuleVersionInfo` struct. The other two parameter are optional, and should be initialised to `NULL` when not required. The other parameters are defined as

```
typedef pointer (*ModuleSetupProc)(pointer, pointer, int *, int
*)
```

```
typedef void (*ModuleTearDownProc)(pointer)
```

```
pointer SetupProc(pointer module, pointer options,

          int *errmaj, int *errmin)
```

When defined, this function is called by the loader after successfully loading a module. module is a handle for the newly loaded module, and maybe used by the `SetupProc` if it calls other loader functions that require a reference to it. The remaining arguments are those that were passed to the `LoadModule()` (or `LoadSubModule()`), and are described above. When the `SetupProc` is successful it must return a non-`NULL` value. The loader checks this, and if it is `NULL` it unloads the module and reports the failure to the caller of `LoadModule()`. If the `SetupProc` does things that need to be undone when the module is unloaded, it should define a `TearDownProc`, and return a pointer that the `TearDownProc` can use to undo what has been done.

When a module is loaded multiple times, the `SetupProc` is called once for each time it is loaded.

```
void TearDownProc(pointer tearDownData)
```

When defined, this function is called when the loader unloads a module. The `tearDownData` parameter is the return value of the `SetupProc()` that was called when the module was loaded. The purpose of this function is to clean up before the module is unloaded (for example, by freeing allocated resources).

17.4 Public Loader Interface

The following is the Loader interface that is available to any part of the server, and may also be used from within modules.

```
pointer LoadSubModule(pointer parent, const char *module,
          const char **subdirlist, const char **patternlist,
          pointer options, const XF86ModReqInfo * modreq,
          int *errmaj, int *errmin)
```

This function is like the `LoadModule()` function described above, except that the module loaded is registered as a child of the calling module. The `parent` parameter is the calling module's handle. Modules loaded with this function are automatically unloaded when the parent module is unloaded. The other difference is that the path parameter may not be specified. The module search path used for modules loaded with this function is the default search path as initialised with `LoaderSetPath()`.

```
void UnloadSubModule(pointer module)
```

This function unloads the module with handle `module`. If that module itself has children, they are also unloaded. It is like `UnloadModule()`, except that it is safe to use for unloading child modules.

```
pointer LoaderSymbol(const char *symbol)
```

This function returns the address of the symbol with name `symbol`. This may be used to locate a module entry point with a known name.

```
char **LoaderlistDirs(const char **subdirlist,
          const char **patternlist)
```

This function returns a `NULL` terminated list of canonical modules names for modules found in the default module search path. The `subdirlist` and `patternlist` parameters are as described above, and can be used to control the locations and names that are searched. If no modules are found, the return value is `NULL`. The returned list should be freed by calling `LoaderFreeDirList()` when it is no longer needed.

```
void LoaderFreeDirList(char **list)
```

This function frees a module list created by `LoaderlistDirs()`.

```
void LoaderReqSymLists(const char **list0, ...)
```

This function allows the registration of required symbols with the loader. It is normally used by a caller of `LoadSubModule()`. If any symbols registered in this way are found to be unresolved when `LoaderCheckUnresolved()` is called then `LoaderCheckUnresolved()` will report a failure. The function takes one or more `NULL` terminated lists of symbols. The end of the argument list is indicated by a `NULL` argument.

```
void LoaderReqSymbols(const char *sym0, ...)
```

This function is like `LoaderReqSymLists()` except that its arguments are symbols rather than lists of symbols. This function is more convenient when single functions are to be registered, especially when the single function might depend on runtime factors. The end of the argument list is indicated by a NULL argument.

`void LoaderRefSymLists(const char **list0, ...)`

This function allows the registration of possibly unresolved symbols with the loader. When `LoaderCheckUnresolved()` is run it won't generate warnings for symbols registered in this way unless they were also registered as required symbols. The function takes one or more NULL terminated lists of symbols. The end of the argument list is indicated by a NULL argument.

`void LoaderRefSymbols(const char *sym0, ...)`

This function is like `LoaderRefSymLists()` except that its arguments are symbols rather than lists of symbols. This function is more convenient when single functions are to be registered, especially when the single function might depend on runtime factors. The end of the argument list is indicated by a NULL argument.

`int LoaderCheckUnresolved(int delayflag)`

This function checks for unresolved symbols. It generates warnings for unresolved symbols that have not been registered with `LoaderRefSymLists()`, and maps them to a dummy function. This behaviour may change in future. If unresolved symbols are found that have been registered with `LoaderReqSymLists()` or `LoaderReqSymbols()` then this function returns a non-zero value. If none of these symbols are unresolved the return value is zero, indicating success.

The `delayflag` parameter should normally be set to `LD_RESOLV_IFDONE`.

```
LoaderErrorMsg(const char *name, const char *modname,

          int errmaj, int errmin)
```

This function prints an error message that includes the text "Failed to load module", the module name `modname`, a message specific to the `errmaj` value, and the value if `errmin`. If `name` is non-NULL, it is printed as an identifying prefix to the message (followed by a ':').

17.5 Special Registration Functions

The loader contains some functions for registering some classes of modules. These may be moved out of the loader at some point.

`void LoadExtension(ExtensionModule *ext)`

This registers the entry points for the extension identified by `ext`. The `ExtensionModule` struct is defined as:

```
typedef struct {
    InitExtension       initFunc;
    char *              name;
    Bool                *disablePtr;
    InitExtension       setupFunc;
} ExtensionModule;
```

`void LoadFont(FontModule *font)`

This registers the entry points for the font rasteriser module identified by `font`. The `FontModule` struct is defined as:

```
typedef struct {
    InitFont    initFunc;
    char *      name;
    pointer     module;
} FontModule;
```

18. Helper Functions

This section describe ‘‘helper’’ functions that video driver might find useful. While video drivers are not required to use any of these to be considered ‘‘compliant’’, the use of appropriate helpers is strongly encouraged to improve the consistency of driver behaviour.

18.1 Functions for printing messages

`ErrorF(const char *format, ...)`

This is the basic function for writing to the error log (typically stderr and/or a log file). Video drivers should usually avoid using this directly in favour of the more specialised functions described below. This function is useful for printing messages while debugging a driver.

`FatalError(const char *format, ...)`

This prints a message and causes the Xserver to abort. It should rarely be used within a video driver, as most error conditions should be flagged by the return values of the driver functions. This allows the higher layers to decide how to proceed. In rare cases, this can be used within a driver if a fatal unexpected condition is found.

`xf86ErrorF(const char *format, ...)`

This is like `ErrorF()`, except that the message is only printed when the Xserver's verbosity level is set to the default (`1`) or higher. It means that the messages are not printed when the server is started with the `-quiet` flag. Typically this function would only be used for continuing messages started with one of the more specialised functions described below.

`xf86ErrorFVerb(int verb, const char *format, ...)`

Like `xf86ErrorF()`, except the minimum verbosity level for which the message is to be printed is given explicitly. Passing a `verb` value of zero means the message is always printed. A value higher than `1` can be used for information would normally not be needed, but which might be useful when diagnosing problems.

`xf86Msg(MessageType type, const char *format, ...)`

This is like `xf86ErrorF()`, except that the message is prefixed with a marker determined by the value of `type`. The marker is used to indicate the type of message (warning, error, probed value, config value, etc). Note the `xf86Verbose` value is ignored for messages of type `X_ERROR`.

The marker values are:

`X_PROBED`

Value was probed.

`X_CONFIG`

Value was given in the config file.

`X_DEFAULT`

Value is a default.

`X_CMDLINE`

Value was given on the command line.

`X_NOTICE`

Notice.

`X_ERROR`

Error message.

`X_WARNING`

Warning message.

`X_INFO`

Informational message.

`X_NONE`

No prefix.

`X_NOT_IMPLEMENTED`

The message relates to functionality that is not yet implemented.

`xf86MsgVerb(MessageType type, int verb, const char *format, ...)`

Like `xf86Msg()`, but with the verbosity level given explicitly.

`xf86DrvMsg(int scrnIndex, MessageType type, const char *format, ...)`

This is like `xf86Msg()` except that the driver's name (the `name` field of the `ScrnInfoRec`) followed by the `scrnIndex` in parentheses is printed following the prefix. This should be used by video drivers in most cases as it clearly indicates which driver/screen the message is for. If `scrnIndex` is negative, this function behaves exactly like `xf86Msg()`.

NOTE: This function can only be used after the `ScrnInfoRec` and its `name` field have been allocated. Normally, this means that it can not be used before the END of the `ChipProbe()` function. Prior to that, use `xf86Msg()`, providing the driver's name explicitly. No screen number can be supplied at that point.

```
xf86DrvMsgVerb(int scrnIndex, MessageType type, int verb,

        const char *format, ...)
```

Like `xf86DrvMsg()`, but with the verbosity level given explicitly.

18.2 Functions for setting values based on command line and config file

`Bool xf86SetDepthBpp(ScrnInfoPtr scrp, int depth, int bpp,`

`int fbbpp, int depth24flags)`

This function sets the `depth`, `pixmapBPP` and `bitsPerPixel` fields of the `ScrnInfoRec`. It also determines the defaults for display-wide attributes and pixmap formats the screen will support, and finds the Display subsection that matches the depth/bpp. This function should normally be called very early from the `ChipPreInit()` function.

It requires that the `confScreen` field of the `ScrnInfoRec` be initialised prior to calling it. This is done by the XFree86 common layer prior to calling `ChipPreInit()`.

The parameters passed are:

`depth`

driver's preferred default depth if no other is given. If zero, use the overall server default.

`bpp`

Same, but for the pixmap bpp.

`fbbpp`

Same, but for the framebuffer bpp.

`depth24flags`

Flags that indicate the level of 24/32bpp support and whether conversion between different framebuffer and pixmap formats is supported. The flags for this argument are defined as follows, and multiple flags may be ORed together:

`NoDepth24Support`

No depth 24 formats supported

`Support24bppFb`

24bpp framebuffer supported

`Support32bppFb`

32bpp framebuffer supported

`SupportConvert24to32`

Can convert 24bpp pixmap to 32bpp fb

`SupportConvert32to24`

Can convert 32bpp pixmap to 24bpp fb

`ForceConvert24to32`

Force 24bpp pixmap to 32bpp fb conversion

`ForceConvert32to24`

Force 32bpp pixmap to 24bpp fb conversion

It uses the command line, config file, and default values in the correct order of precedence to determine the depth and bpp values. It is up to the driver to check the results to see that it supports them. If not the `ChipPreInit()` function should return `FALSE`.

If only one of depth/bpp is given, the other is set to a reasonable (and consistent) default.

If a driver finds that the initial `depth24flags` it uses later results in a fb format that requires more video memory than is available it may call this function a second time with a different `depth24flags` setting.

On success, the return value is `TRUE`. On failure it prints an error message and returns `FALSE`.

The following fields of the `ScrnInfoRec` are initialised by this function:

`depth`, `bitsPerPixel`, `display`, `imageByteOrder`, `bitmapScanlinePad`, `bitmapScanlineUnit`, `bitmapBitOrder`, `numFormats`, `formats`, `fbFormat`.

`void xf86PrintDepthBpp(scrnInfoPtr scrp)`

This function can be used to print out the depth and bpp settings. It should be called after the final call to `xf86SetDepthBpp()`.

`Bool xf86SetWeight(ScrnInfoPtr scrp, rgb weight, rgb mask)`

This function sets the `weight`, `mask`, `offset` and `rgbBits` fields of the `ScrnInfoRec`. It would normally be called fairly early in the `ChipPreInit()` function for depths > 8bpp.

It requires that the `depth` and `display` fields of the `ScrnInfoRec` be initialised prior to calling it.

The parameters passed are:

`weight`

driver's preferred default weight if no other is given. If zero, use the overall server default.

`mask`

Same, but for mask.

It uses the command line, config file, and default values in the correct order of precedence to determine the weight value. It derives the mask and offset values from the weight and the defaults. It is up to the driver to check the results to see that it supports them. If not the `ChipPreInit()` function should return `FALSE`.

On success, this function prints a message showing the weight values selected, and returns `TRUE`.

On failure it prints an error message and returns `FALSE`.

The following fields of the `ScrnInfoRec` are initialised by this function:

`weight`, `mask`, `offset`.

`Bool xf86SetDefaultVisual(ScrnInfoPtr scrp, int visual)`

This function sets the `defaultVisual` field of the `ScrnInfoRec`. It would normally be called fairly early from the `ChipPreInit()` function.

It requires that the `depth` and `display` fields of the `ScrnInfoRec` be initialised prior to calling it.

The parameters passed are:

`visual`

driver's preferred default visual if no other is given. If `-1`, use the overall server default.

It uses the command line, config file, and default values in the correct order of precedence to determine the default visual value. It is up to the driver to check the result to see that it supports it. If not the `ChipPreInit()` function should return `FALSE`.

On success, this function prints a message showing the default visual selected, and returns `TRUE`.

On failure it prints an error message and returns `FALSE`.

`Bool xf86SetGamma(ScrnInfoPtr scrp, Gamma gamma)`

This function sets the `gamma` field of the `ScrnInfoRec`. It would normally be called fairly early from the `ChipPreInit()` function in cases where the driver supports gamma correction.

It requires that the `monitor` field of the `ScrnInfoRec` be initialised prior to calling it.

The parameters passed are:
`gamma`

driver's preferred default gamma if no other is given. If zero (< `0.01`), use the overall server default.

It uses the command line, config file, and default values in the correct order of precedence to determine the gamma value. It is up to the driver to check the results to see that it supports them. If not the `ChipPreInit()` function should return `FALSE`.

On success, this function prints a message showing the gamma value selected, and returns `TRUE`.

On failure it prints an error message and returns `FALSE`.

`void xf86SetDpi(ScrnInfoPtr pScrn, int x, int y)`

This function sets the `xDpi` and `yDpi` fields of the `ScrnInfoRec`. The driver can specify preferred defaults by setting `x` and `y` to non-zero values. The `-dpi` command line option overrides all other settings. Otherwise, if the **DisplaySize** entry is present in the screen's **Monitor** config file section, it is used together with the virtual size to calculate the dpi values. This function should be called after all the mode resolution has been done.

`void xf86SetBlackWhitePixels(ScrnInfoPtr pScrn)`

This functions sets the `blackPixel` and `whitePixel` fields of the `Scrn-`

`InfoRec` according to whether or not the `-flipPixels` command line options is present.

`const char *xf86GetVisualName(int visual)`

Returns a printable string with the visual name matching the numerical visual class provided. If the value is outside the range of valid visual classes, `NULL` is returned.

18.3 Primary Mode functions

The primary mode helper functions are those which would normally be used by a driver, unless it has unusual requirements which cannot be catered for the by the helpers.

```
int xf86ValidateModes(ScrnInfoPtr scrp, DisplayModePtr availModes,
          char **modeNames, ClockRangePtr clockRanges,
          int *linePitches, int minPitch, int maxPitch,
          int pitchInc, int minHeight, int maxHeight,
          int virtualX, int virtualY,
          unsigned long apertureSize,
          LookupModeFlags strategy)
```

This function basically selects the set of modes to use based on those available and the various constraints. It also sets some other related parameters. It is normally called near the end of the `ChipPreInit()` function.

The parameters passed to the function are:

`availModes`

List of modes available for the monitor.

`modeNames`

List of mode names that the screen is requesting.

`clockRanges`

A list of clock ranges allowed by the driver. Each range includes whether interlaced or multiscan modes are supported for that range. See below for more on `clockRanges`.

`linePitches`

List of line pitches supported by the driver. This is optional and should be `NULL` when not used.

`minPitch`

Minimum line pitch supported by the driver. This must be supplied when `linePitches` is `NULL`, and is ignored otherwise.

`maxPitch`

Maximum line pitch supported by the driver. This is required when `minPitch` is required.

`pitchInc`

Granularity of horizontal pitch values as supported by the chipset. This is expressed in bits. This must be supplied.

`minHeight`

minimum virtual height allowed. If zero, no limit is imposed.

`maxHeight`

maximum virtual height allowed. If zero, no limit is imposed.

`virtualX`

If greater than zero, this is the virtual width value that will be used. Otherwise, the virtual width is chosen to be the smallest that can accommodate the modes selected.

`virtualY`

If greater than zero, this is the virtual height value that will be used. Otherwise, the virtual height is chosen to be the smallest that can accommodate the modes selected.

`apertureSize`

The size (in bytes) of the aperture used to access video memory.

`strategy`

The strategy to use when choosing from multiple modes with the same name. The options are:

`LOOKUP_DEFAULT`

???

`LOOKUP_BEST_REFRESH`

mode with best refresh rate

`LOOKUP_CLOSEST_CLOCK`

mode with closest matching clock

`LOOKUP_LIST_ORDER`

first usable mode in list

The following options can also be combined (OR'ed) with one of the above:

`LOOKUP_CLKDIV2`

Allow halved clocks

`LOOKUP_OPTIONAL_TOLERANCES`

Allow missing horizontal sync and/or vertical refresh ranges in the XF86Config Monitor section

`LOOKUP_OPTIONAL_TOLERANCES` should only be specified when the driver can ensure all modes it generates can sync on, or at least not damage, the monitor or digital flat panel. Horizontal sync and/or vertical refresh ranges specified by the user will still be honoured (and acted upon).

This function requires that the following fields of the `ScrnInfoRec` are initialised prior to calling it:

`clock[]`

List of discrete clocks (when non-programmable)

`numClocks`

Number of discrete clocks (when non-programmable)

`progClock`

Whether the clock is programmable or not

`monitor`

Pointer to the applicable XF86Config monitor section

`fdFormat`

Format of the screen buffer

`videoRam`

total video memory size (in bytes)

`maxHValue`

Maximum horizontal timing value allowed

`maxVValue`

Maximum vertical timing value allowed

`xInc`

Horizontal timing increment in pixels (defaults to 8)

This function fills in the following `ScrnInfoRec` fields:

`modePool`

A subset of the modes available to the monitor which are compatible with the driver.

`modes`

One mode entry for each of the requested modes, with the status field of each filled in to indicate if the mode has been accepted or not. This list of modes is a circular list.

`virtualX`

The resulting virtual width.

`virtualY`

The resulting virtual height.

`displayWidth`

The resulting line pitch.

`virtualFrom`

Where the virtual size was determined from.

The first stage of this function checks that the `virtualX` and `virtualY` values supplied (if greater than zero) are consistent with the line pitch and `maxHeight` limitations. If not, an error message is printed, and the return value is `-1`.

The second stage sets up the mode pool, eliminating immediately any modes that exceed the driver's line pitch limits, and also the virtual width and height limits (if greater than zero). For each mode removed an informational message is printed at verbosity level `2`. If the mode pool ends up being empty, a warning message is printed, and the return value is `0`.

The final stage is to lookup each mode name, and fill in the remaining parameters. If an error condition is encountered, a message is printed, and the return value is `-1`. Otherwise, the return value is the number of valid modes found (`0` if none are found).

Even if the supplied mode names include duplicates, no two names will ever match the same mode. Furthermore, if the supplied mode names do not yield a valid mode (including the case where no names are passed at all), the function will continue looking through the mode pool until it finds a mode that survives all checks, or until the mode pool is exhausted.

A message is only printed by this function when a fundamental problem is found. It is intended that this function may be called more than once if there is more than one set of constraints that the driver can work within.

If this function returns `-1`, the `ChipPreInit()` function should return `FALSE`.

`clockRanges` is a linked list of clock ranges allowed by the driver. If a mode doesn't fit in any of the defined `clockRanges`, it is rejected. The first `clockRange` that matches all requirements is used. This structure needs to be initialized to NULL when allocated.

`clockRanges` contains the following fields:

`minClock`

`maxClock`

> The lower and upper mode clock bounds for which the rest of the `clockRange` parameters apply. Since these are the mode clocks, they are not scaled with the `ClockMulFactor` and `ClockDivFactor`. It is up to the driver to adjust these values if they depend on the clock scaling factors.

`clockIndex`

> (not used yet) `-1` for programmable clocks

`interlaceAllowed`

> `TRUE` if interlacing is allowed for this range

`doubleScanAllowed`

> `TRUE` if doublescan or multiscan is allowed for this range

`ClockMulFactor`

`ClockDivFactor`

> Scaling factors that are applied to the mode clocks ONLY before selecting a clock index (when there is no programmable clock) or a `SynthClock` value. This is useful for drivers that support pixel multiplexing or that need to scale the clocks because of hardware restrictions (like sending 24bpp data to an 8 bit RAMDAC using a tripled clock).
>
> Note that these parameters describe what must be done to the mode clock to achieve the data transport clock between graphics controller and RAMDAC. For example for `2:1` pixel multiplexing, two pixels are sent to the RAMDAC on each clock. This allows the RAMDAC clock to be half of the actual pixel clock. Hence, `ClockMulFactor=1` and `ClockDivFactor=2`.

This means that the clock used for clock selection (ie, determining the correct clock index from the list of discrete clocks) or for the `SynthClock` field in case of a programmable clock is: `(mode->Clock * ClockMulFactor) / ClockDivFactor`.

`PrivFlags`

This field is copied into the `mode->PrivFlags` field when this `clockRange` is selected by `xf86ValidateModes()`. It allows the driver to find out what clock range was selected, so it knows it needs to set up pixel multiplexing or any other range-dependent feature. This field is purely driver-defined: it may contain flag bits, an index or anything else (as long as it is an `INT`).

Note that the `mode->SynthClock` field is always filled in by `xf86ValidateModes()`: it will contain the “data transport clock”, which is the clock that will have to be programmed in the chip when it has a programmable clock, or the clock that will be picked from the clocks list when it is not a programmable one. Thus:

```
mode->SynthClock =

          (mode->Clock * ClockMulFactor) / ClockDivFac-
tor
```

`void xf86PruneDriverModes(ScrnInfoPtr scrp)`

This function deletes modes in the modes field of the `ScrnInfoRec` that have been marked as invalid. This is normally run after having run `xf86ValidateModes()` for the last time. For each mode that is deleted, a warning message is printed out indicating the reason for it being deleted.

`void xf86SetCrtcForModes(ScrnInfoPtr scrp, int adjustFlags)`

This function fills in the `Crtc*` fields for all the modes in the `modes` field of the `ScrnInfoRec`. The `adjustFlags` parameter determines how the vertical CRTC values are scaled for interlaced modes. They are halved if it is `INTERLACE_HALVE_V`. The vertical CRTC values are doubled for doublescan modes, and are further multiplied by the `VScan` value.

This function is normally called after calling `xf86PruneDriverModes()`.

`void xf86PrintModes(ScrnInfoPtr scrp)`

This function prints out the virtual size setting, and the line pitch being used. It also prints out two lines for each mode being used. The first line includes the mode’s pixel clock, horizontal sync rate, refresh rate, and whether it is interlaced, doublescanned and/or multi-scanned. The second line is the mode’s Modeline.

This function is normally called after calling `xf86SetCrtcForModes()`.

18.4 Secondary Mode functions

The secondary mode helper functions are functions which are normally used by the primary mode helper functions, and which are not normally called directly by a driver. If a driver has unusual requirements and needs to do its own mode validation, it might be able to make use of some of these secondary mode helper functions.

```
int xf86GetNearestClock(ScrnInfoPtr scrp, int freq, Bool allow-
Div2,

          int *divider)
```

This function returns the index of the closest clock to the frequency `freq` given (in kHz). It assumes that the number of clocks is greater than zero. It requires that the `numClocks` and `clock` fields of the `ScrnInfoRec` are initialised. The `allowDiv2` field determines if the clocks can be halved. The `*divider` return value indicates whether clock division is used when determining the clock returned.

This function is only for non-programmable clocks.

```
const char *xf86ModeStatusToString(ModeStatus status)
```

This function converts the `status` value to a descriptive printable string.

```
ModeStatus xf86LookupMode(ScrnInfoPtr scrp, DisplayModePtr
modep,

          ClockRangePtr clockRanges, LookupModeFlags strategy)
```

This function takes a pointer to a mode with the name filled in, and looks for a mode in the `modePool` list which matches. The parameters of the matching mode are filled in to `*modep`. The `clockRanges` and `strategy` parameters are as for the `xf86ValidateModes()` function above.

This function requires the `modePool`, `clock[]`, `numClocks` and `progClock` fields of the `ScrnInfoRec` to be initialised before being called. The return value is `MODE_OK` if a mode was found. Otherwise it indicates why a matching mode could not be found.

```
ModeStatus xf86InitialCheckModeForDriver(ScrnInfoPtr scrp,

          DisplayModePtr mode, ClockRangePtr clockRanges,

          LookupModeFlags strategy, int maxPitch,

          int virtualX, int virtualY)
```

This function checks the passed mode against some basic driver constraints. Apart from the ones passed explicitly, the `maxHValue` and `maxVValue` fields of the `ScrnInfoRec` are also used. If the `ValidMode` field of the `ScrnInfoRec` is set, that function is also called to check the mode. Next, the mode is checked against the monitor's constraints.

If the mode is consistent with all constraints, the return value is `MODE_OK`. Otherwise the return value indicates which constraint wasn't met.

```
void xf86DeleteMode(DisplayModePtr *modeList, DisplayModePtr
mode)
```

This function deletes the `mode` given from the `modeList`. It never prints any messages, so it is up to the caller to print a message if required.

18.5 Functions for handling strings and tokens

Tables associating strings and numerical tokens combined with the following functions provide a compact way of handling strings from the config file, and for converting tokens into printable strings. The table data structure is:

```
typedef struct {
    int                 token;
    const char *        name;
} SymTabRec, *SymTabPtr;
```

A table is an initialised array of `SymTabRec`. The tokens must be non-negative integers. Multiple names may be mapped to a single token. The table is terminated with an element with a `token` value of `-1` and `NULL` for the `name`.

`const char *xf86TokenToString(SymTabPtr table, int token)`

> This function returns the first string in `table` that matches `token`. If no match is found, `NULL` is returned (NOTE, older versions of this function would return the string "unknown" when no match is found).

`int xf86StringToToken(SymTabPtr table, const char *string)`

> This function returns the first token in `table` that matches `string`. The `xf86NameCmp()` function is used to determine the match. If no match is found, `-1` is returned.

18.6 Functions for finding which config file entries to use

These functions can be used to select the appropriate config file entries that match the detected hardware. They are described above in the *Probe* (section 5.8, page 8) and *Available Functions* (section 9.3, page 24) sections.

18.7 Probing discrete clocks on old hardware

The `xf86GetClocks()` function may be used to assist in finding the discrete pixel clock values on older hardware.

```
void xf86GetClocks(ScrnInfoPtr pScrn, int num,
          Bool (*ClockFunc)(ScrnInfoPtr, int),
          void (*ProtectRegs)(ScrnInfoPtr, Bool),
          void (*BlankScreen)(ScrnInfoPtr, Bool),
          int vertsyncreg, int maskval, int knownclkindex,
          int knownclkvalue)
```

This function uses a comparative sampling method to measure the discrete pixel clock values. The number of discrete clocks to measure is given by `num`. `clockFunc` is a function that selects the `n`'th clock. It should also save or restore any state affected by programming the clocks when the index passed is `CLK_REG_SAVE` or `CLK_REG_RESTORE`. `ProtectRegs` is a function that does whatever is required to protect the hardware state while selecting a new clock. `BlankScreen` is a function that blanks the screen. `vertsyncreg` and `maskval` are the register and bitmask to check for the presence of vertical sync pulses. `knownclkindex` and `knownclkvalue` are the index and value of a known clock. These are the known references on which the comparative measurements are based. The number of clocks probed is set in `pScrn->numClocks`, and the probed clocks are set in the `pScrn->clock[]` array. All of the clock values are in units of kHz.

```
void xf86ShowClocks(ScrnInfoPtr scrp, MessageType from)
```

Print out the pixel clocks `scrp->clock[]`. `from` indicates whether the clocks were probed or from the config file.

18.8 Other helper functions

```
Bool xf86IsUnblank(int mode)
```

Returns `TRUE` when the screen saver mode specified by `mode` requires the screen be unblanked, and `FALSE` otherwise. The screen saver modes that require blanking are `SCREEN_SAVER_ON` and `SCREEN_SAVER_CYCLE`, and the screen saver modes that require unblanking are `SCREEN_SAVER_OFF` and `SCREEN_SAVER_FORCER`. Drivers may call this helper from their `SaveScreen()` function to interpret the screen saver modes.

19. The vgahw module

The vgahw modules provides an interface for saving, restoring and programming the standard VGA registers, and for handling VGA colourmaps.

19.1 Data Structures

The public data structures used by the vgahw module are `vgaRegRec` and `vgaHWRec`. They are defined in `vgaHW.h.`

19.2 General vgahw Functions

`Bool vgaHWGetHWRec(ScrnInfoPtr pScrn)`

This function allocates a `vgaHWRec` structure, and hooks it into the `ScrnInfoRec`'s `privates`. Like all information hooked into the `privates`, it is persistent, and only needs to be allocated once per screen. This function should normally be called from the driver's `ChipPreInit()` function. The `vgaHWRec` is zero-allocated, and the following fields are explicitly initialised:

`ModeReg.DAC[]`

initialised with a default colourmap

`ModeReg.Attribute[0x11]`

initialised with the default overscan index

`ShowOverscan`

initialised according to the "ShowOverscan" option

`paletteEnabled`

initialised to FALSE

`cmapSaved`

initialised to FALSE

`pScrn`

initialised to pScrn

In addition to the above, `vgaHWSetStdFuncs()` is called to initialise the register access function fields with the standard VGA set of functions.

Once allocated, a pointer to the `vgaHWRec` can be obtained from the `ScrnInfoPtr` with the `VGAHWPTR(pScrn)` macro.

`void vgaHWFreeHWRec(ScrnInfoPtr pScrn)`

This function frees a `vgaHWRec` structure. It should be called from a driver's `ChipFreeScreen()` function.

`Bool vgaHWSetRegCounts(ScrnInfoPtr pScrn, int numCRTC,`

`int numSequencer, int numGraphics, int numAttribute)`

This function allows the number of CRTC, Sequencer, Graphics and Attribute registers to be changed. This makes it possible for extended registers to be saved and restored with `vgaHWSave()` and `vgaHWRestore()`. This function should be called after a `vgaHWRec` has been allocated with `vgaHWGetHWRec()`. The default values are defined in `vgaHW.h` as follows:

```
#define VGA_NUM_CRTC 25
#define VGA_NUM_SEQ   5
```

```
#define VGA_NUM_GFX   9
#define VGA_NUM_ATTR 21
```

`Bool vgaHWCopyReg(vgaRegPtr dst, vgaRegPtr src)`

This function copies the contents of the VGA saved registers in `src` to `dst`. Note that it isn't possible to simply do this with `memcpy()` (or similar). This function returns `TRUE` unless there is a problem allocating space for the `CRTC` and related fields in `dst`.

`void vgaHWSetStdFuncs(vgaHWPtr hwp)`

This function initialises the register access function fields of `hwp` with the standard VGA set of functions. This is called by `vgaHWGetHWRec()`, so there is usually no need to call this explicitly. The register access functions are described below. If the registers are shadowed in some other port I/O space (for example a PCI I/O region), these functions can be used to access the shadowed registers if `hwp->PIOOffset` is initialised with `offset`, calculated in such a way that when the standard VGA I/O port value is added to it the correct offset into the PIO area results. This value is initialised to zero in `vgaHWGetHWRec()`. (Note: the PIOOffset functionality is present in XFree86 4.1.0 and later.)

`void vgaHWSetMmioFuncs(vgaHWPtr hwp, CARD8 *base, int offset)`

This function initialised the register access function fields of hwp with a generic MMIO set of functions. `hwp->MMIOBase` is initialised with `base`, which must be the virtual address that the start of MMIO area is mapped to. `hwp->MMIOOffset` is initialised with `offset`, which must be calculated in such a way that when the standard VGA I/O port value is added to it the correct offset into the MMIO area results. That means that these functions are only suitable when the VGA I/O ports are made available in a direct mapping to the MMIO space. If that is not the case, the driver will need to provide its own register access functions. The register access functions are described below.

`Bool vgaHWMapMem(ScrnInfoPtr pScrn)`

This function maps the VGA memory window. It requires that the `vgaHWRec` be allocated. If a driver requires non-default `MapPhys` or `MapSize` settings (the physical location and size of the VGA memory window) then those fields of the `vgaHWRec` must be initialised before calling this function. Otherwise, this function initialiases the default values of `0xA0000` for `MapPhys` and `(64 * 1024)` for `MapSize`. This function must be called before attempting to save or restore the VGA state. If the driver doesn't call it explicitly, the `vgaHWSave()` and `vgaHWRestore()` functions may call it if they need to access the VGA memory (in which case they will also call `vgaHWUnmapMem()` to unmap the VGA memory before exiting).

`void vgaHWUnmapMem(ScrnInfoPtr pScrn)`

This function unmaps the VGA memory window. It must only be called after the memory has been mapped. The `Base` field of the `vgaHWRec` field is set to `NULL` to indicate that the memory is no longer mapped.

`void vgaHWGetIOBase(vgaHWPtr hwp)`

This function initialises the `IOBase` field of the `vgaHWRec`. This function must be called before using any other functions that access the video hardware.

A macro `VGAHW_GET_IOBASE()` is also available in `vgaHW.h` that returns the I/O base, and this may be used when the vgahw module is not loaded (for example, in the `ChipProbe()` function).

`void vgaHWUnlock(vgaHWPtr hwp)`

This function unlocks the VGA `CRTC[0-7]` registers, and must be called before attempting to write to those registers.

`void vgaHWLock(vgaHWPtr hwp)`

This function locks the VGA `CRTC[0-7]` registers.

`void vgaHWEnable(vgaHWPtr hwp)`

This function enables the VGA subsystem. (Note, this function is present in XFree86 4.1.0 and later.).

`void vgaHWDisable(vgaHWPtr hwp)`

This function disables the VGA subsystem. (Note, this function is present in XFree86 4.1.0 and later.).

`void vgaHWSave(ScrnInfoPtr pScrn, vgaRegPtr save, int flags)`

This function saves the VGA state. The state is written to the `vgaRegRec` pointed to by `save`. `flags` is set to one or more of the following flags ORed together:

`VGA_SR_MODE`

the mode setting registers are saved

`VGA_SR_FONTS`

the text mode font/text data is saved

`VGA_SR_CMAP`

the colourmap (LUT) is saved

`VGA_SR_ALL`

all of the above are saved

The `vgaHWRec` and its `IOBase` fields must be initialised before this function is called. If `VGA_SR_FONTS` is set in `flags`, the VGA memory window must be mapped. If it isn't then `vgaHWMapMem()` will be called to map it, and `vgaHWUnmapMem()` will be called to unmap it afterwards. `vgaHWSave()` uses the three functions below in the order `vgaHWSaveColormap()`, `vgaHWSaveMode()`, `vgaHWSaveFonts()` to carry out the different save phases. It is undecided at this stage whether they will remain part of the vgahw module's public interface or not.

`void vgaHWSaveMode(ScrnInfoPtr pScrn, vgaRegPtr save)`

This function saves the VGA mode registers. They are saved to the `vgaRegRec` pointed to by `save`. The registers saved are:

`MiscOut`

`CRTC[0-0x18]`

`Attribute[0-0x14]`

`Graphics[0-8]`

`Sequencer[0-4]`

The number of registers actually saved may be modified by a prior call to `vgaHWSetRegCounts()`.

`void vgaHWSaveFonts(ScrnInfoPtr pScrn, vgaRegPtr save)`

This function saves the text mode font and text data held in the video memory. If called while in a graphics mode, no save is done. The VGA memory window must be mapped with `vgaHWMapMem()` before to calling this function.

On some platforms, one or more of the font/text plane saves may be no-ops. This is the case when the platform's VC driver already takes care of this.

`void vgaHWSaveColormap(ScrnInfoPtr pScrn, vgaRegPtr save)`

This function saves the VGA colourmap (LUT). Before saving it, it attempts to verify that the colourmap is readable. In rare cases where it isn't readable, a default colourmap is saved instead.

`void vgaHWRestore(ScrnInfoPtr pScrn, vgaRegPtr restore, int flags)`

This function programs the VGA state. The state programmed is that contained in the `vgaRegRec` pointed to by `restore`. `flags` is the same as described above for the `vgaHWSave()` function.

The `vgaHWRec` and its `IOBase` fields must be initialised before this function is called. If `VGA_SR_FONTS` is set in `flags`, the VGA memory window must be mapped. If it isn't then `vgaHWMapMem()` will be called to map it, and `vgaHWUnmapMem()` will be called to unmap it afterwards. `vgaHWRestore()` uses the three functions below in the order `vgaHWRestoreFonts()`, `vgaHWRestoreMode()`, `vgaHWRestoreColormap()` to carry out the different restore phases. It is undecided at this stage whether they will remain part of the vgahw module's public interface or not.

`void vgaHWRestoreMode(ScrnInfoPtr pScrn, vgaRegPtr restore)`

This function restores the VGA mode registers. They are restored from the data in the `vgaRegRec` pointed to by `restore`. The registers restored are:

`MiscOut`

`CRTC[0-0x18]`

`Attribute[0-0x14]`

`Graphics[0-8]`

`Sequencer[0-4]`

The number of registers actually restored may be modified by a prior call to `vgaHWSetRegCounts()`.

`void vgaHWRestoreFonts(ScrnInfoPtr pScrn, vgaRegPtr restore)`

This function restores the text mode font and text data to the video memory. The VGA memory window must be mapped with `vgaHWMapMem()` before to calling this function.

On some platforms, one or more of the font/text plane restores may be no-ops. This is the case when the platform's VC driver already takes care of this.

`void vgaHWRestoreColormap(ScrnInfoPtr pScrn, vgaRegPtr restore)`

This function restores the VGA colourmap (LUT).

`void vgaHWInit(ScrnInfoPtr pScrn, DisplayModePtr mode)`

This function fills in the `vgaHWRec`'s `ModeReg` field with the values appropriate for programming the given video mode. It requires that the `ScrnInfoRec`'s `depth` field is initialised, which determines how the registers are programmed.

`void vgaHWSeqReset(vgaHWPtr hwp, Bool start)`

Do a VGA sequencer reset. If start is `TRUE`, the reset is started. If start is `FALSE`, the reset is ended.

`void vgaHWProtect(ScrnInfoPtr pScrn, Bool on)`

This function protects VGA registers and memory from corruption during loads. It is typically called with on set to `TRUE` before programming, and with on set to `FALSE` after programming.

`Bool vgaHWSaveScreen(ScreenPtr pScreen, int mode)`

This function blanks and unblanks the screen. It is blanked when `mode` is `SCREEN_SAVER_ON` or `SCREEN_SAVER_CYCLE`, and unblanked when `mode` is `SCREEN_SAVER_OFF` or `SCREEN_SAVER_FORCER`.

`void vgaHWBlankScreen(ScrnInfoPtr pScrn, Bool on)`

This function blanks and unblanks the screen. It is blanked when `on` is `FALSE`, and unblanked when `on` is `TRUE`. This function is provided for use in cases where the `ScrnInfoRec` can't be derived from the `ScreenRec` (while probing for clocks, for example).

19.3 VGA Colormap Functions

The vgahw module uses the standard colormap support (see the *Colormap Handling* (section 13., page 41) section. This is initialised with the following function:

`Bool vgaHWHandleColormaps(ScreenPtr pScreen)`

19.4 VGA Register Access Functions

The vgahw module abstracts access to the standard VGA registers by using a set of functions held in the `vgaHWRec`. When the `vgaHWRec` is created these function pointers are initialised with the set of standard VGA I/O register access functions. In addition to these, the vgahw module includes a basic set of MMIO register access functions, and the `vgaHWRec` function pointers can be initialised to these by calling the `vgaHWSetMmioFuncs()` function described above. Some drivers/platforms may require a different set of functions for VGA access. The access functions are described here.

`void writeCrtc(vgaHWPtr hwp, CARD8 index, CARD8 value)`

Write `value` to CRTC register `index`.

`CARD8 readCrtc(vgaHWPtr hwp, CARD8 index)`

Return the value read from CRTC register `index`.

`void writeGr(vgaHWPtr hwp, CARD8 index, CARD8 value)`

Write `value` to Graphics Controller register `index`.

`CARD8 readGR(vgaHWPtr hwp, CARD8 index)`

Return the value read from Graphics Controller register `index`.

`void writeSeq(vgaHWPtr hwp, CARD8 index, CARD8, value)`

Write `value` to Sequencer register `index`.

`CARD8 readSeq(vgaHWPtr hwp, CARD8 index)`

Return the value read from Sequencer register `index`.

`void writeAttr(vgaHWPtr hwp, CARD8 index, CARD8, value)`

Write `value` to Attribute Controller register `index`. When writing out the index value this function should set bit 5 (`0x20`) according to the setting of `hwp->paletteEnabled` in order to preserve the palette access state. It should be cleared when `hwp->paletteEnabled` is `TRUE` and set when it is `FALSE`.

`CARD8 readAttr(vgaHWPtr hwp, CARD8 index)`

Return the value read from Attribute Controller register `index`. When writing out the index value this function should set bit 5 (`0x20`) according to the setting of `hwp->paletteEnabled` in order to preserve the palette access state. It should be cleared when `hwp->paletteEnabled` is `TRUE` and set when it is `FALSE`.

`void writeMiscOut(vgaHWPtr hwp, CARD8 value)`

Write '`value`' to the Miscellaneous Output register.

`CARD8 readMiscOut(vgwHWPtr hwp)`

Return the value read from the Miscellaneous Output register.

`void enablePalette(vgaHWPtr hwp)`

Clear the palette address source bit in the Attribute Controller index register and set `hwp->paletteEnabled` to `TRUE`.

`void disablePalette(vgaHWPtr hwp)`

Set the palette address source bit in the Attribute Controller index register and set `hwp->paletteEnabled` to `FALSE`.

`void writeDacMask(vgaHWPtr hwp, CARD8 value)`

Write `value` to the DAC Mask register.

`CARD8 readDacMask(vgaHWptr hwp)`

Return the value read from the DAC Mask register.

`void writeDacReadAddress(vgaHWPtr hwp, CARD8 value)`

Write `value` to the DAC Read Address register.

`void writeDacWriteAddress(vgaHWPtr hwp, CARD8 value)`

Write `value` to the DAC Write Address register.

`void writeDacData(vgaHWPtr hwp, CARD8 value)`

Write `value` to the DAC Data register.

`CARD8 readDacData(vgaHWptr hwp)`

Return the value read from the DAC Data register.

`CARD8 readEnable(vgaHWptr hwp)`

Return the value read from the VGA Enable register. (Note: This function is present in XFree86 4.1.0 and later.)

`void writeEnable(vgaHWPtr hwp, CARD8 value)`

Write `value` to the VGA Enable register. (Note: This function is present in XFree86 4.1.0 and later.)

20. Some notes about writing a driver

NOTE: some parts of this are not up to date

The following is an outline for writing a basic unaccelerated driver for a PCI video card with a linear mapped framebuffer, and which has a VGA core. It is includes some general information that is relevant to most drivers (even those which don't fit that basic description).

The information here is based on the initial conversion of the Matrox Millennium driver to the "new design". For a fleshing out and sample implementation of some of the bits outlined here, refer to that driver. Note that this is an example only. The approach used here will not be appropriate for all drivers.

Each driver must reserve a unique driver name, and a string that is used to prefix all of its externally visible symbols. This is to avoid name space clashes when loading multiple drivers. The examples here are for the "ZZZ" driver, which uses the "ZZZ" or "zzz" prefix for its externally visible symbols.

20.1 Include files

All drivers normally include the following headers:

```
"xf86.h"

"xf86_OSproc.h"

"xf86_ansic.h"

"xf86Resources.h"
```

Wherever inb/outb (and related things) are used the following should be included:

```
"compiler.h"
```

Note: in drivers, this must be included after `"xf86_ansic.h"`.

Drivers that need to access PCI vendor/device definitions need this:

```
"xf86PciInfo.h"
```

Drivers that need to access the PCI config space need this:

```
"xf86Pci.h"
```

Drivers using the mi banking wrapper need:

```
"mibank.h"
```

Drivers that initialise a SW cursor need this:

```
"mipointer.h"
```

All drivers implementing backing store need this:

```
"mibstore.h"
```

All drivers using the mi colourmap code need this:

```
"micmap.h"
```

If a driver uses the vgahw module, it needs this:

```
"vgaHW.h"
```

Drivers supporting VGA or Hercules monochrome screens need:

```
"xf1bpp.h"
```

Drivers supporting VGA or EGC 16-colour screens need:

```
"xf4bpp.h"
```

Drivers using cfb need:

```
#define PSZ 8

#include "cfb.h"

#undef PSZ
```

Drivers supporting bpp 16, 24 or 32 with cfb need one or more of:

```
"cfb16.h"

"cfb24.h"

"cfb32.h"
```

The driver's own header file:

```
"zzz.h"
```

Drivers must NOT include the following:

"xf86Priv.h"

"xf86Privstr.h"

"xf86_libc.h"

"xf86_OSlib.h"

"Xos.h"

any OS header

20.2 Data structures and initialisation

- The following macros should be defined:

```
#define VERSION <version-as-an-int>
#define ZZZ_NAME "ZZZ"         /* the name used to prefix messages */
#define ZZZ_DRIVER_NAME "zzz"  /* the driver name as used in config file */
#define ZZZ_MAJOR_VERSION <int>
#define ZZZ_MINOR_VERSION <int>
#define ZZZ_PATCHLEVEL    <int>
```

  NOTE: `ZZZ_DRIVER_NAME` should match the name of the driver module without things like the "lib" prefix, the "_drv" suffix or filename extensions.

- A DriverRec must be defined, which includes the functions required at the pre-probe phase. The name of this DriverRec must be an upper-case version of ZZZ_DRIVER_NAME (for the purposes of static linking).

```
DriverRec ZZZ = {
    VERSION,
    ZZZ_DRIVER_NAME,
    ZZZIdentify,
    ZZZProbe,
    ZZZAvailableOptions,
    NULL,
    0
};
```

- Define list of supported chips and their matching ID:

```
static SymTabRec ZZZChipsets[] = {
    { PCI_CHIP_ZZZ1234, "zzz1234a" },
    { PCI_CHIP_ZZZ5678, "zzz5678a" },
    { -1,               NULL }
};
```

  The token field may be any integer value that the driver may use to uniquely identify the supported chipsets. For drivers that support only PCI devices using the PCI device IDs might be a natural choice, but this isn't mandatory. For drivers that support both PCI and other devices (like ISA), some other ID should probably used. When other IDs are used as the tokens it is recommended that the names be defined as an `enum` type.

- If the driver uses the `xf86MatchPciInstances()` helper (recommended for drivers that support PCI cards) a list that maps PCI IDs to chip IDs and fixed resources must be defined:

```
static PciChipsets ZZZPciChipsets[] = {
    { PCI_CHIP_ZZZ1234, PCI_CHIP_ZZZ1234, RES_SHARED_VGA },
    { PCI_CHIP_ZZZ5678, PCI_CHIP_ZZZ5678, RES_SHARED_VGA },
    { -1,               -1,               RES_UNDEFINED }
}
```

- Define the `XF86ModuleVersionInfo` struct for the driver. This is required for the dynamically loaded version:

```
#ifdef XFree86LOADER
static XF86ModuleVersionInfo zzzVersRec =
{
    "zzz",
    MODULEVENDORSTRING,
    MODINFOSTRING1,
    MODINFOSTRING2,
    XF86_VERSION_CURRENT,
    ZZZ_MAJOR_VERSION, ZZZ_MINOR_VERSION, ZZZ_PATCHLEVEL,
    ABI_CLASS_VIDEODRV,
    ABI_VIDEODRV_VERSION,
    MOD_CLASS_VIDEODRV,
    {0,0,0,0}
};
#endif
```

- Define a data structure to hold the driver's screen-specific data. This must be used instead of global variables. This would be defined in the `"zzz.h"` file, something like:

```
typedef struct {
    type1  field1;
    type2  field2;
    int    fooHack;
    Bool   pciRetry;
    Bool   noAccel;
    Bool   hwCursor;
    CloseScreenProcPtr CloseScreen;
    OptionInfoPtr Options;
    ...
} ZZZRec, *ZZZPtr;
```

- Define the list of config file Options that the driver accepts. For consistency between drivers those in the list of "standard" options should be used where appropriate before inventing new options.

```
typedef enum {
    OPTION_FOO_HACK,
    OPTION_PCI_RETRY,
    OPTION_HW_CURSOR,
    OPTION_NOACCEL
} ZZZOpts;

static const OptionInfoRec ZZZOptions[] = {
  { OPTION_FOO_HACK,  "FooHack",   OPTV_INTEGER, {0}, FALSE },
  { OPTION_PCI_RETRY, "PciRetry",  OPTV_BOOLEAN, {0}, FALSE },
  { OPTION_HW_CURSOR, "HWcursor",  OPTV_BOOLEAN, {0}, FALSE },
  { OPTION_NOACCEL,   "NoAccel",   OPTV_BOOLEAN, {0}, FALSE },
  { -1,               NULL,        OPTV_NONE,    {0}, FALSE }
};
```

20.3 Functions

20.3.1 SetupProc

For dynamically loaded modules, a `ModuleData` variable is required. It is should be the name of the driver prepended to "ModuleData". A `Setup()` function is also required, which calls `xf86AddDriver()` to add the driver to the main list of drivers.

```
#ifdef XFree86LOADER

static MODULESETUPPROTO(mgaSetup);

XF86ModuleData zzzModuleData = { &zzzVersRec, zzzSetup, NULL };

static pointer
zzzSetup(pointer module, pointer opts, int *errmaj, int *errmin)
{
    static Bool setupDone = FALSE;

    /* This module should be loaded only once, but check to be sure. */

    if (!setupDone) {
        /*
         * Modules that this driver always requires may be loaded
         * here  by calling LoadSubModule().
         */

        setupDone = TRUE;
        xf86AddDriver(&MGA, module, 0);

        /*
         * The return value must be non-NULL on success even though
         * there is no TearDownProc.
         */
        return (pointer)1;
    } else {
        if (errmaj) *errmaj = LDR_ONCEONLY;
        return NULL;
    }
}
#endif
```

20.3.2 GetRec, FreeRec

A function is usually required to allocate the driver's screen-specific data structure and hook it into the `ScrnInfoRec`'s `driverPrivate` field. The `ScrnInfoRec`'s `driverPrivate` is initialised to `NULL`, so it is easy to check if the initialisation has already been done. After allocating it, initialise the fields. By using `xnfcalloc()` to do the allocation it is zeroed, and if the allocation fails the server exits.

NOTE: When allocating structures from inside the driver which are defined on the common level it is important to initialize the structure to zero. Only this guarantees that the server remains source compatible to future changes in common level structures.

```
static Bool
ZZZGetRec(ScrnInfoPtr pScrn)
{
    if (pScrn->driverPrivate != NULL)
        return TRUE;
    pScrn->driverPrivate = xnfcalloc(sizeof(ZZZRec), 1);
    /* Initialise as required */
    ...
    return TRUE;
}
```

Define a macro in `"zzz.h"` which gets a pointer to the `ZZZRec` when given `pScrn`:

```
#define ZZZPTR(p) ((ZZZPtr)((p)->driverPrivate))
```

Define a function to free the above, setting it to `NULL` once it has been freed:

```
static void
ZZZFreeRec(ScrnInfoPtr pScrn)
{
    if (pScrn->driverPrivate == NULL)
        return;
    xfree(pScrn->driverPrivate);
    pScrn->driverPrivate = NULL;
}
```

20.3.3 Identify

Define the `Identify()` function. It is run before the Probe, and typically prints out an identifying message, which might include the chipsets it supports. This function is mandatory:

```
static void
ZZZIdentify(int flags)
{
    xf86PrintChipsets(ZZZ_NAME, "driver for ZZZ Tech chipsets",
                      ZZZChipsets);
}
```

20.3.4 Probe

Define the `Probe()` function. The purpose of this is to find all instances of the hardware that the driver supports, and for the ones not already claimed by another driver, claim the slot, and allocate a `ScrnInfoRec`. This should be a minimal probe, and it should under no circumstances leave the state of the hardware changed. Because a device is found, don't assume that it will be used. Don't do any initialisations other than the required `ScrnInfoRec` initialisations. Don't allocate any new data structures.

This function is mandatory.

NOTE: The `xf86DrvMsg()` functions cannot be used from the Probe.

```
static Bool
ZZZProbe(DriverPtr drv, int flags)
{
    Bool foundScreen = FALSE;
    int numDevSections, numUsed;
    GDevPtr *devSections;
    int *usedChips;
    int i;

    /*
     * Find the config file Device sections that match this
     * driver, and return if there are none.
     */
    if ((numDevSections = xf86MatchDevice(ZZZ_DRIVER_NAME,
                                          &devSections)) <= 0) {
        return FALSE;
    }

    /*
     * Since this is a PCI card, "probing" just amounts to checking
     * the PCI data that the server has already collected.  If there
     * is none, return.
     *
     * Although the config file is allowed to override things, it
     * is reasonable to not allow it to override the detection
     * of no PCI video cards.
     *
     * The provided xf86MatchPciInstances() helper takes care of
     * the details.
     */
    /* test if PCI bus present */
    if (xf86GetPciVideoInfo()) {

        numUsed = xf86MatchPciInstances(ZZZ_NAME, PCI_VENDOR_ZZZ,
                        ZZZChipsets, ZZZPciChipsets, devSections,
                        numDevSections, drv, &usedChips);

        for (i = 0; i < numUsed; i++) {
            ScrnInfoPtr pScrn = NULL;
            if ((pScrn = xf86ConfigPciEntity(pScrn, flags, usedChips[i],
                                             ZZZPciChipsets, NULL, NULL,
                                             NULL, NULL, NULL))) {
               /* Allocate a ScrnInfoRec */
               pScrn->driverVersion = VERSION;
               pScrn->driverName    = ZZZ_DRIVER_NAME;
               pScrn->name          = ZZZ_NAME;
               pScrn->Probe         = ZZZProbe;
               pScrn->PreInit       = ZZZPreInit;
               pScrn->ScreenInit    = ZZZScreenInit;
               pScrn->SwitchMode    = ZZZSwitchMode;
               pScrn->AdjustFrame   = ZZZAdjustFrame;
               pScrn->EnterVT       = ZZZEnterVT;
               pScrn->LeaveVT       = ZZZLeaveVT;
               pScrn->FreeScreen    = ZZZFreeScreen;
               pScrn->ValidMode     = ZZZValidMode;
               foundScreen = TRUE;
               /* add screen to entity */
           }
        }
        xfree(usedChips);
    }

#ifdef HAS_ISA_DEVS
    /*
     * If the driver supports ISA hardware, the following block
```

```
     * can be included too.
     */
    numUsed = xf86MatchIsaInstances(ZZZ_NAME, ZZZChipsets,
                             ZZZIsaChipsets, drv, ZZZFindIsaDevice,
                             devSections, numDevSections, &usedChips);
    for (i = 0; i < numUsed; i++) {
        ScrnInfoPtr pScrn = NULL;
        if ((pScrn = xf86ConfigIsaEntity(pScrn, flags, usedChips[i],
                                      ZZZIsaChipsets, NULL, NULL, NULL,
                                      NULL, NULL))) {
            pScrn->driverVersion = VERSION;
            pScrn->driverName    = ZZZ_DRIVER_NAME;
            pScrn->name          = ZZZ_NAME;
            pScrn->Probe         = ZZZProbe;
            pScrn->PreInit       = ZZZPreInit;
            pScrn->ScreenInit    = ZZZScreenInit;
            pScrn->SwitchMode    = ZZZSwitchMode;
            pScrn->AdjustFrame   = ZZZAdjustFrame;
            pScrn->EnterVT       = ZZZEnterVT;
            pScrn->LeaveVT       = ZZZLeaveVT;
            pScrn->FreeScreen    = ZZZFreeScreen;
            pScrn->ValidMode     = ZZZValidMode;
            foundScreen = TRUE;
        }
    }
    xfree(usedChips);
#endif /* HAS_ISA_DEVS */

    xfree(devSections);
    return foundScreen;
```

20.3.5 AvailableOptions

Define the `AvailableOptions()` function. The purpose of this is to return the available driver options back to the -configure option, so that an XF86Config file can be built and the user can see which options are available for them to use.

20.3.6 PreInit

Define the `PreInit()` function. The purpose of this is to find all the information required to determine if the configuration is usable, and to initialise those parts of the `ScrnInfoRec` that can be set once at the beginning of the first server generation. The information should be found in the least intrusive way possible.

This function is mandatory.

NOTES:

1. The `PreInit()` function is only called once during the life of the X server (at the start of the first generation).
2. Data allocated here must be of the type that persists for the life of the X server. This means that data that hooks into the `ScrnInfoRec`'s `privates` field should be allocated here, but data that hooks into the `ScreenRec`'s `devPrivates` field should not be allocated here. The `driverPrivate` field should also be allocated here.
3. Although the `ScrnInfoRec` has been allocated before this function is called, the `ScreenRec` has not been allocated. That means that things requiring it cannot be used in this function.
4. Very little of the `ScrnInfoRec` has been initialised when this function is called. It is important to get the order of doing things right in this function.

```
static Bool
ZZZPreInit(ScrnInfoPtr pScrn, int flags)
{
    /* Fill in the monitor field */
    pScrn->monitor = pScrn->confScreen->monitor;

    /*
     * If using the vgahw module, it will typically be loaded
     * here by calling xf86LoadSubModule(pScrn, "vgahw");
     */

    /*
     * Set the depth/bpp.  Use the globally preferred depth/bpp.  If the
     * driver has special default depth/bpp requirements, the defaults should
     * be specified here explicitly.
     * We support both 24bpp and 32bpp framebuffer layouts.
     * This sets pScrn->display also.
     */
    if (!xf86SetDepthBpp(pScrn, 0, 0, 0,
                         Support24bppFb | Support32bppFb)) {
        return FALSE;
    } else {
        if (depth/bpp isn't one we support) {
            print error message;
            return FALSE;
        }
    }
    /* Print out the depth/bpp that was set */
    xf86PrintDepthBpp(pScrn);

    /* Set bits per RGB for 8bpp */
    if (pScrn->depth <= 8) {
        /* Take into account a dac_6_bit option here */
        pScrn->rgbBits = 6 or 8;
    }

    /*
     * xf86SetWeight() and xf86SetDefaultVisual() must be called
     * after pScrn->display is initialised.
     */

    /* Set weight/mask/offset for depth > 8 */
    if (pScrn->depth > 8) {
        if (!xf86SetWeight(pScrn, defaultWeight, defaultMask)) {
            return FALSE;
        } else {
            if (weight isn't one we support) {
                print error message;
                return FALSE;
            }
        }
    }

    /* Set the default visual. */
    if (!xf86SetDefaultVisual(pScrn, -1)) {
        return FALSE;
    } else {
        if (visual isn't one we support) {
            print error message;
            return FALSE;
        }
    }

    /* If the driver supports gamma correction, set the gamma. */
    if (!xf86SetGamma(pScrn, default_gamma)) {
```

```
        return FALSE;
    }

    /* This driver uses a programmable clock */
    pScrn->progClock = TRUE;

    /* Allocate the ZZZRec driverPrivate */
    if (!ZZZGetRec(pScrn)) {
        return FALSE;
    }

    pZzz = ZZZPTR(pScrn);

    /* Collect all of the option flags (fill in pScrn->options) */
    xf86CollectOptions(pScrn, NULL);

    /*
     * Process the options based on the information in ZZZOptions.
     * The results are written to pZzz->Options.  If all of the options
     * processing is done within this function a local variable "options"
     * can be used instead of pZzz->Options.
     */
    if (!(pZzz->Options = xalloc(sizeof(ZZZOptions))))
        return FALSE;
    (void)memcpy(pZzz->Options, ZZZOptions, sizeof(ZZZOptions));
    xf86ProcessOptions(pScrn->scrnIndex, pScrn->options, pZzz->Options);

    /*
     * Set various fields of ScrnInfoRec and/or ZZZRec based on
     * the options found.
     */
    from = X_DEFAULT;
    pZzz->hwCursor = FALSE;
    if (xf86IsOptionSet(pZzz->Options, OPTION_HW_CURSOR)) {
        from = X_CONFIG;
        pZzz->hwCursor = TRUE;
    }
    xf86DrvMsg(pScrn->scrnIndex, from, "Using %s cursor\n",
               pZzz->hwCursor ? "HW" : "SW");
    if (xf86IsOptionSet(pZzz->Options, OPTION_NOACCEL)) {
        pZzz->noAccel = TRUE;
        xf86DrvMsg(pScrn->scrnIndex, X_CONFIG,
                   "Acceleration disabled\n");
    } else {
        pZzz->noAccel = FALSE;
    }
    if (xf86IsOptionSet(pZzz->Options, OPTION_PCI_RETRY)) {
        pZzz->UsePCIRetry = TRUE;
        xf86DrvMsg(pScrn->scrnIndex, X_CONFIG, "PCI retry enabled\n");
    }
    pZzz->fooHack = 0;
    if (xf86GetOptValInteger(pZzz->Options, OPTION_FOO_HACK,
                             &pZzz->fooHack)) {
        xf86DrvMsg(pScrn->scrnIndex, X_CONFIG, "Foo Hack set to %d\n",
                   pZzz->fooHack);
    }

    /*
     * Find the PCI slot(s) that this screen claimed in the probe.
     * In this case, exactly one is expected, so complain otherwise.
     * Note in this case we're not interested in the card types so
     * that parameter is set to NULL.
     */
    if ((i = xf86GetPciInfoForScreen(pScrn->scrnIndex, &pciList, NULL))
        != 1) {
```

```
        print error message;
        ZZZFreeRec(pScrn);
        if (i > 0)
            xfree(pciList);
        return FALSE;
    }
    /* Note that pciList should be freed below when no longer needed */

    /*
     * Determine the chipset, allowing config file chipset and
     * chipid values to override the probed information.  The config
     * chipset value has precedence over its chipid value if both
     * are present.
     *
     * It isn't necessary to fill in pScrn->chipset if the driver
     * keeps track of the chipset in its ZZZRec.
     */

    ...

    /*
     * Determine video memory, fb base address, I/O addresses, etc,
     * allowing the config file to override probed values.
     *
     * Set the appropriate pScrn fields (videoRam is probably the
     * most important one that other code might require), and
     * print out the settings.
     */

    ...

    /* Initialise a clockRanges list. */

    ...

    /* Set any other chipset specific things in the ZZZRec */

    ...

    /* Select valid modes from those available */

    i = xf86ValidateModes(pScrn, pScrn->monitor->Modes,
                          pScrn->display->modes, clockRanges,
                          NULL, minPitch, maxPitch, rounding,
                          minHeight, maxHeight,
                          pScrn->display->virtualX,
                          pScrn->display->virtualY,
                          pScrn->videoRam * 1024,
                          LOOKUP_BEST_REFRESH);
    if (i == -1) {
        ZZZFreeRec(pScrn);
        return FALSE;
    }

    /* Prune the modes marked as invalid */

    xf86PruneDriverModes(pScrn);

    /* If no valid modes, return */

    if (i == 0 || pScrn->modes == NULL) {
        print error message;
        ZZZFreeRec(pScrn);
        return FALSE;
    }
```

```
    /*
     * Initialise the CRTC fields for the modes.  This driver expects
     * vertical values to be halved for interlaced modes.
     */
    xf86SetCrtcForModes(pScrn, INTERLACE_HALVE_V);

    /* Set the current mode to the first in the list. */
    pScrn->currentMode = pScrn->modes;

    /* Print the list of modes being used. */
    xf86PrintModes(pScrn);

    /* Set the DPI */
    xf86SetDpi(pScrn, 0, 0);

    /* Load bpp-specific modules */
    switch (pScrn->bitsPerPixel) {
    case 1:
        mod = "xf1bpp";
        break;
    case 4:
        mod = "xf4bpp";
        break;
    case 8:
        mod = "cfb";
        break;
    case 16:
        mod = "cfb16";
        break;
    case 24:
        mod = "cfb24";
        break;
    case 32:
        mod = "cfb32";
        break;
    }
    if (mod && !xf86LoadSubModule(pScrn, mod))
        ZZZFreeRec(pScrn);
        return FALSE;

    /* Load XAA if needed */
    if (!pZzz->noAccel || pZzz->hwCursor)
        if (!xf86LoadSubModule(pScrn, "xaa")) {
            ZZZFreeRec(pScrn);
            return FALSE;
        }

    /* Done */
    return TRUE;
}
```

20.3.7 MapMem, UnmapMem

Define functions to map and unmap the video memory and any other memory apertures required. These functions are not mandatory, but it is often useful to have such functions.

```
static Bool
ZZZMapMem(ScrnInfoPtr pScrn)
{
    /* Call xf86MapPciMem() to map each PCI memory area */
    ...
    return TRUE or FALSE;
}

static Bool
ZZZUnmapMem(ScrnInfoPtr pScrn)
{
    /* Call xf86UnMapVidMem() to unmap each memory area */
    ...
    return TRUE or FALSE;
}
```

20.3.8 Save, Restore

Define functions to save and restore the original video state. These functions are not mandatory, but are often useful.

```
static void
ZZZSave(ScrnInfoPtr pScrn)
{
    /*
     * Save state into per-screen data structures.
     * If using the vgahw module, vgaHWSave will typically be
     * called here.
     */
    ...
}

static void
ZZZRestore(ScrnInfoPtr pScrn)
{
    /*
     * Restore state from per-screen data structures.
     * If using the vgahw module, vgaHWRestore will typically be
     * called here.
     */
    ...
}
```

20.3.9 ModeInit

Define a function to initialise a new video mode. This function isn't mandatory, but is often useful.

```
static Bool
ZZZModeInit(ScrnInfoPtr pScrn, DisplayModePtr mode)
{
    /*
     * Program a video mode.  If using the vgahw module,
     * vgaHWInit and vgaRestore will typically be called here.
     * Once up to the point where there can't be a failure
     * set pScrn->vtSema to TRUE.
     */
    ...
}
```

20.3.10 ScreenInit

Define the `ScreenInit()` function. This is called at the start of each server generation, and should fill in as much of the `ScreenRec` as possible as well as any other data that is initialised once per generation. It should initialise the framebuffer layers it is using, and initialise the initial video mode.

This function is mandatory.

NOTE: The `ScreenRec` (`pScreen`) is passed to this driver, but it and the `ScrnInfoRecs` are not yet hooked into each other. This means that in this function, and functions it calls, one cannot be found from the other.

```
static Bool
ZZZScreenInit(int scrnIndex, ScreenPtr pScreen, int argc, char **argv)
{
    /* Get the ScrnInfoRec */
    pScrn = xf86Screens[pScreen->myNum];

    /*
     * If using the vgahw module, its data structures and related
     * things are typically initialised/mapped here.
     */

    /* Save the current video state */
    ZZZSave(pScrn);

    /* Initialise the first mode */
    ZZZModeInit(pScrn, pScrn->currentMode);

    /* Set the viewport if supported */

    ZZZAdjustFrame(scrnIndex, pScrn->frameX0, pScrn->frameY0, 0);

    /*
     * Setup the screen's visuals, and initialise the framebuffer
     * code.
     */

    /* Reset the visual list */
    miClearVisualTypes();

    /*
     * Setup the visuals supported.  This driver only supports
     * TrueColor for bpp > 8, so the default set of visuals isn't
     * acceptable.  To deal with this, call miSetVisualTypes with
     * the appropriate visual mask.
     */

    if (pScrn->bitsPerPixel > 8) {
        if (!miSetVisualTypes(pScrn->depth, TrueColorMask,
                              pScrn->rgbBits, pScrn->defaultVisual))
            return FALSE;
    } else {
        if (!miSetVisualTypes(pScrn->depth,
                              miGetDefaultVisualMask(pScrn->depth),
                              pScrn->rgbBits, pScrn->defaultVisual))
            return FALSE;
    }

    /*
     * Initialise the framebuffer.
     */

    switch (pScrn->bitsPerPixel) {
```

```
case 1:
    ret = xf1bppScreenInit(pScreen, FbBase,
                           pScrn->virtualX, pScrn->virtualY,
                           pScrn->xDpi, pScrn->yDpi,
                           pScrn->displayWidth);
    break;
case 4:
    ret = xf4bppScreenInit(pScreen, FbBase,
                           pScrn->virtualX, pScrn->virtualY,
                           pScrn->xDpi, pScrn->yDpi,
                           pScrn->displayWidth);
    break;
case 8:
    ret = cfbScreenInit(pScreen, FbBase,
                        pScrn->virtualX, pScrn->virtualY,
                        pScrn->xDpi, pScrn->yDpi,
                        pScrn->displayWidth);
    break;
case 16:
    ret = cfb16ScreenInit(pScreen, FbBase,
                          pScrn->virtualX, pScrn->virtualY,
                          pScrn->xDpi, pScrn->yDpi,
                          pScrn->displayWidth);
    break;
case 24:
    ret = cfb24ScreenInit(pScreen, FbBase,
                          pScrn->virtualX, pScrn->virtualY,
                          pScrn->xDpi, pScrn->yDpi,
                          pScrn->displayWidth);
    break;
case 32:
    ret = cfb32ScreenInit(pScreen, FbBase,
                          pScrn->virtualX, pScrn->virtualY,
                          pScrn->xDpi, pScrn->yDpi,
                          pScrn->displayWidth);
    break;
default:
    print a message about an internal error;
    ret = FALSE;
    break;
}

if (!ret)
    return FALSE;

/* Override the default mask/offset settings */
if (pScrn->bitsPerPixel > 8) {
    for (i = 0, visual = pScreen->visuals;
         i < pScreen->numVisuals; i++, visual++) {
        if ((visual->class | DynamicClass) == DirectColor) {
            visual->offsetRed = pScrn->offset.red;
            visual->offsetGreen = pScrn->offset.green;
            visual->offsetBlue = pScrn->offset.blue;
            visual->redMask = pScrn->mask.red;
            visual->greenMask = pScrn->mask.green;
            visual->blueMask = pScrn->mask.blue;
        }
    }
}

/*
 * If banking is needed, initialise an miBankInfoRec (defined in
 * "mibank.h"), and call miInitializeBanking().
 */
if (!miInitializeBanking(pScreen, pScrn->virtualX, pScrn->virtualY,
```

```
                                        pScrn->displayWidth, pBankInfo))
            return FALSE;

        /*
         * If backing store is to be supported (as is usually the case),
         * initialise it.
         */
        miInitializeBackingStore(pScreen);

        /*
         * Set initial black & white colourmap indices.
         */
        xf86SetBlackWhitePixels(pScreen);

        /*
         * Install colourmap functions.  If using the vgahw module,
         * vgaHandleColormaps would usually be called here.
         */

        ...

        /*
         * Initialise cursor functions.  This example is for the mi
         * software cursor.
         */
        miDCInitialize(pScreen, xf86GetPointerScreenFuncs());

        /* Initialise the default colourmap */
        switch (pScrn->depth) {
        case 1:
            if (!xf1bppCreateDefColormap(pScreen))
                return FALSE;
            break;
        case 4:
            if (!xf4bppCreateDefColormap(pScreen))
                return FALSE;
            break;
        default:
            if (!cfbCreateDefColormap(pScreen))
                return FALSE;
            break;
        }

        /*
         * Wrap the CloseScreen vector and set SaveScreen.
         */
        ZZZPTR(pScrn)->CloseScreen = pScreen->CloseScreen;
        pScreen->CloseScreen = ZZZCloseScreen;
        pScreen->SaveScreen = ZZZSaveScreen;

        /* Report any unused options (only for the first generation) */
        if (serverGeneration == 1) {
            xf86ShowUnusedOptions(pScrn->scrnIndex, pScrn->options);
        }

        /* Done */
        return TRUE;
    }
```

20.3.11 SwitchMode

Define the `SwitchMode()` function if mode switching is supported by the driver.

```
static Bool
ZZZSwitchMode(int scrnIndex, DisplayModePtr mode, int flags)
{
    return ZZZModeInit(xf86Screens[scrnIndex], mode);
}
```

20.3.12 AdjustFrame

Define the `AdjustFrame()` function if the driver supports this.

```
static void
ZZZAdjustFrame(int scrnIndex, int x, int y, int flags)
{
    /* Adjust the viewport */
}
```

20.3.13 EnterVT, LeaveVT

Define the `EnterVT()` and `LeaveVT()` functions.

These functions are mandatory.

```
static Bool
ZZZEnterVT(int scrnIndex, int flags)
{
    ScrnInfoPtr pScrn = xf86Screens[scrnIndex];
    return ZZZModeInit(pScrn, pScrn->currentMode);
}

static void
ZZZLeaveVT(int scrnIndex, int flags)
{
    ScrnInfoPtr pScrn = xf86Screens[scrnIndex];
    ZZZRestore(pScrn);
}
```

20.3.14 CloseScreen

Define the `CloseScreen()` function:

This function is mandatory. Note that it unwraps the previously wrapped `pScreen->CloseScreen`, and finishes by calling it.

```
static Bool
ZZZCloseScreen(int scrnIndex, ScreenPtr pScreen)
{
    ScrnInfoPtr pScrn = xf86Screens[scrnIndex];
    if (pScrn->vtSema) {
        ZZZRestore(pScrn);
        ZZZUnmapMem(pScrn);
    }
    pScrn->vtSema = FALSE;
    pScreen->CloseScreen = ZZZPTR(pScrn)->CloseScreen;
    return (*pScreen->CloseScreen)(scrnIndex, pScreen);
}
```

20.3.15 SaveScreen

Define the `SaveScreen()` function (the screen blanking function). When using the vgahw module, this will typically be:

```
static Bool
ZZZSaveScreen(ScreenPtr pScreen, int mode)
{
    return vgaHWSaveScreen(pScreen, mode);
}
```

This function is mandatory. Before modifying any hardware register directly this function needs to make sure that the Xserver is active by checking if `pScrn` is non-NULL and for `pScrn->vtSema == TRUE`.

20.3.16 FreeScreen

Define the `FreeScreen()` function. This function is optional. It should be defined if the `ScrnInfoRec driverPrivate` field is used so that it can be freed when a screen is deleted by the common layer for reasons possibly beyond the driver's control. This function is not used in during normal (error free) operation. The per-generation data is freed by the `CloseScreen()` function.

```
static void
ZZZFreeScreen(int scrnIndex, int flags)
{
    /*
     * If the vgahw module is used vgaHWFreeHWRec() would be called
     * here.
     */
    ZZZFreeRec(xf86Screens[scrnIndex]);
}
```

CONTENTS

`$XFree86: xc/programs/Xserver/hw/xfree86/doc/sgml/DESIGN.sgml,v 1.54 2003/12/19 20:38:57 dawes E`