Multimedia Driver Support in the FreeBSD Operating System

James S. Lowe
University of Wisconsin-Milwaukee
College of Engineering and Applied Science
james@cs.uwm.edu
http://www.uwm.edu/~james

Luigi Rizzo
Universita' di Pisa
Dip. di Ingegneria dell'Informazione
l.rizzo@iet.unipi.it
http://www.iet.unipi.it/~luigi

Abstract

In this paper we will discuss the motivation, design, implementation issues, and applications for the audio and the video acquisition drivers which are available in FreeBSD [1]. Both systems have been designed from scratch, with special attention paid to the effective transfer and synchronization of data between the hardware and advanced applications.

The main focus in the design of the audio driver was for supporting multimedia applications, which often require full duplex operation and have strict synchronization requirements. This suggested the definition of a new software interface, simpler to use than the previously existing OSS (Voxware) driver. The driver is backward compatible with the OSS API due to the large base of applications that use this API.

The video acquisition driver focuses on providing flexible access to the acquisition device and the ability to suit the needs of different applications, from simple TV viewers to video conferencing programs. The driver provides memory mapped access to the capture buffer, flexible support for synchronization with the application, and supports the various capture formats and scaling capabilities that are available from the video acquisition hardware.

1 Introduction

Multimedia applications are becoming more and more popular everyday due to the increasing performance of workstations and networks. The ability to run highly demanding compression algorithms in software has promoted the development of advanced applications which deliver multimedia streams, requiring synchronization between the application and the hardware.

Acquisition and rendering of multimedia streams requires special peripherals such as, audio CODECs[1] and frame grabbers, as well as, adequate support in the operating system. In fact, CODECs and frame grabbers are relatively simple devices, and in principle they could be easily supported in an operating system using the conventional `read()` and `write()` system calls, plus a small number of `ioctl()` calls to configure the special features of the devices. However, synchronization is a fundamental requirement with many advanced applications, so the internal structure of the driver, and the interface it exports, must support these requirements in a flexible and efficient way. Furthermore, video streams are highly demanding in terms of bandwidth, so the conventional `read()/write()` interface might not be the most efficient or effective way to transfer data.

In recent years, multimedia support in the FreeBSD operating system has received a lot of attention, namely, the addition of support for full duplex audio cards and high performance video grabbers communicating through the PCI bus. Most of these systems have been designed from scratch. In this process, we had the significant advantage of already knowing which applications would likely use these devices, and what features were required. As a consequence, we could use this information to provide efficient support for the applications, without missing some important features, and without cluttering the design with features that are of no practical use.

For the video acquisition driver, we were essentially free to design our software interface from scratch. For the audio driver, the existence of a relatively

[1] The term CODEC comes from a hybrid of the words COmpressor and DECompressor.

large base of applications forced us to implement a compatibility interface. We decided to look at compatibility issues only after the architecture of the driver was designed, so that these issues did not influence our design.

The paper is structured as follows. In Section 2 we present the audio driver, starting with a brief description of audio hardware, followed by a discussion of the various features implemented by the driver, and a brief discussion of our design choices. We discuss the same topics in Section 3 for video acquisition support. Finally, Section 4 discusses the implementation status of the drivers presented in this paper and Section 5 contains our conclusions.

2 Audio

The structure of a typical audio device is shown in Figure 1 with two logically independent systems[2] providing support for audio record and playback.

In the record section, analog sources are routed through a *mixer* to the Analog to Digital Converter (ADC), whose output is stored into a buffer. The hardware dictates the native resolution (8 or 16 bits), data formats (linear or μ-law), channels (mono/stereo) and sampling rate. The software implements the buffer and may additionally implement some data format conversion before passing data to the application program.

The playback section uses a buffer to hold data that is sent to the Digital to Analog Converter (DAC). The output of the DAC is fed to a mixer which combines together other analog sources and sends them to the appropriate outputs. Similar to recording, the operating system software implements the buffering strategy and possible format conversions to supplement the native features supplied by the hardware.

The DAC and the ADC are usually part of the same physical device referred to as the *CODEC*. The transfer of samples between the CODEC and the buffers supplied by the operating system can be done in several ways. Most of the time, the audio device uses a DMA channel which is present on the

[2] Except for cases where hardware limitations make the system share resources thus preventing full duplex operation or odd mixer characteristics.

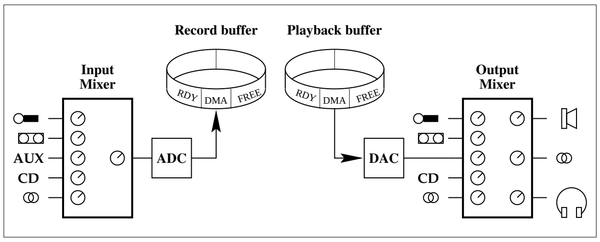

Figure 1: A model of a typical full duplex audio device. On the left is the record (or input) section, on the right the playback (or output) section.

system's mainboard (as in the case of Intel-based PC's). In some other cases, the audio device has an internal DMA engine which acquires control of the bus and performs the data transfer (devices on a PCI bus generally operate in this way). As an alternative, the audio device could have some internal memory where data is buffered (e.g. under control of a specialized processor), and the main processor has only to transfer blocks of data at given intervals, possibly using programmed I/O.

In some cases the audio device has an on board Digital Signal Processor (DSP) which can be used to run data compression algorithms, offloading the main processor from this task. This is, in general, a useful because some algorithms (e.g. GSM or MPEG encoding/decoding) are expensive and it is cheaper to run them on a DSP (which has become a commodity device because it is extensively used in cellular phones and other digital audio devices).

There are large variations in the capabilities of the mixers as well. Some simple devices just have one input and one output channel, directly connected to the ADC and the DAC, respectively, with no volume controls. The majority of audio devices for PC's have a simple multiplexer with only a master volume control on the input section, and a full featured mixer with independent volume controls for the various channels on the output section. Newer devices have separate full-featured mixers on both the input and output paths containing a large selection of sources and destinations along with the ability of performing miscellaneous functions such as swapping left and right channels, muting sources, etc.

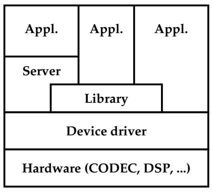

Figure 2: Possible structure of applications using the audio device. Application access to the driver can be direct, via a library, or through a server.

2.1 Audio Device Initialization

The "device driver" layer in Figure 2, is responsible for the transfer of data between the applications and the audio hardware. Prior to using the audio device for data transfer, applications will need to acquire the capabilities of the device (supported sampling rates, data formats, number of channels, full duplex operation, and any other device-specific "features"[3]), and set the desired data formats. These operations are usually done by means of `ioctl()` calls, whose name and format change from system to system. Although some approaches (e.g. those where all parameters are read or set at once, using a single call) appear to be more elegant and efficient than others, there are in practice no differences because this is generally a one-time operation.

Another one-time operation, for audio devices that include a programmable DSP, might be the downloading of appropriate firmware to the DSP to perform the required function (e.g. running some compression or decompression algorithm). Dedicated software interfaces are often used for this purpose, and efficiency is generally not a primary concern.

Finally, appropriate `ioctl()` calls are necessary to control the mixers which are present in the signal paths. This is an area which would really benefit from some standardization effort, given the significant differences in capabilities of these devices. However, the control of mixer devices is conceptually simple because the requested operation (e.g. setting a volume or selecting an input source) generally takes place in real time, and the only significant issue is to have an interface which can ease the porting of software to different systems.

[3]There are significant differences among audio cards. As an example, some CODECS can only work in full duplex under some constraints; others have bugs that are triggered by certain operating modes, and so on. The driver can block erroneous requests, but the only way to make good use of the available hardware is to have the driver provide a unique identifier for the actual hardware and applications (or libraries) can map these identifiers to a list of features and adapt to them.

2.2 Audio Access Granularity

The natural representation of sound is a stream of data. Our driver emulates this natural representation by supplying the application with the appropriate tools to manipulate a circular queue rather than requiring the application to collect and manipulate the data in blocks.

The audio driver has two modes of operation, *character* and *block* mode. In *character* mode, the device produces a stream of bytes, and `select()` returns when one byte can be transferred. To enter the *block mode* (not to be confused with *blocking mode*, which is an orthogonal feature), the `AIOSSIZE ioctl()` is used to specify the granularity to be used for the `select()` operation. The latter will return successfully only when *at least* a whole block of data can be transferred. `AIOSSIZE ioctl()` call can modify the requested value if it is not an acceptable one, and it returns the block size in use. A block size of 0 or 1 will set the device driver into character mode.

We want to point out that the `AIOSSIZE` function only specifies the behavior of the `select()` call; both `read()` and `write()` retain the usual byte granularity. We found this to be a necessary feature because it permits applications to control the data transfer rate with fine granularity, rather than being forced to use multiples of the block size. For robustness reasons, the user shouldn't make further assumptions on the behavior of `read()` and `write()`. In particular, it should not be assumed that they always transfer the requested amount of data, or that they always work on multiples of the block size. Similarly, the user should make no assumptions on the internal operation of the driver (e.g. that the size of DMA transfers equals the value specified with `AIOSSIZE`).

2.2.1 Non-Blocking I/O for Audio

Traditionally, device drivers provide support for non blocking I/O. This operating mode can be selected by issuing the `FIONBIO ioctl()`. In non-blocking mode, read and write operations will never block, at the price of possibly returning a short transfer count[4]. Non blocking reads are possible, even in blocking mode, by invoking the `FIONREAD ioctl()` first, and then reading no more than the amount of available data.

There is no standard function which is equivalent to `FIONREAD` for write operations. In our driver we have implemented such a function, called `AIONWRITE`[5], which returns the free space in the playback buffer. A write of this many bytes will not block, even if blocking mode is selected. In our implementation, both `FIONREAD` and `AIONWRITE` track the status of a DMA transfer.

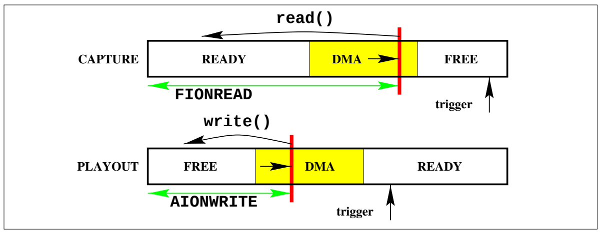

Figure 3: A view of the record and playback buffers, with the indication of events which trigger the actions requested with `AIOSYNC`. The thick vertical bar indicates the current position of the DMA pointer. Also indicated are the effect of `read()` and `write()`, and the values returned by `FIONREAD` and `AIONWRITE`.

2.3 Audio Synchronization

It is important for some applications to know the exact status of the internal buffers in the device driver, both in terms of ready and free space, because some other activities could be synchronized with the audio streams. As an example, a player program may want to avoid the condition that the playback buffer becomes empty in the middle of operation; or, a telephone-like application might need to control the amount of data buffered, in order to limit the end-to-end latency.

In principle, when the data transfer occurs at a constant nominal rate, the amount of data buffered can be derived by using a real time clock. In practice, this method is imprecise because of deviations between the nominal and the actual sample rate, clock drifts, or buffer overflows/underflows which cause samples to be missed. Furthermore, variable-rate compression schemes, or dynamically-changing buffer sizes, might render the use of timers for determining queue occupation completely useless.

`FIONREAD` and `AIONWRITE` only provide limited information on the status of buffers, and are used mainly when the application would like to avoid blocking on the device. We would also like to have alternative mechanisms which either notify a process (e.g. using a Unix `signal`) or wake it up when a specified event occurs. Figure 3 shows in detail the record and playback buffers. The boundary between the FREE and READY regions, which we call the *current DMA pointer*, moves with time. Applications may be interested when the current DMA pointer reaches a given position relative to either the beginning or the end of the buffer and be notified when this event occurs, as well as, knowing the exact position of the current DMA pointer relative to either end of the buffer.

To support these functionalities we have introduced a single new `ioctl()`, `AIOSYNC`, which takes the specification of an event (the current DMA pointer reaching a given position in either buffer) and an action to execute when the event occurs. The event can be specified relative to either end of the buffer, while the action can be any of the following (in all cases, upon return, the current DMA pointer will be reported, relative to the same buffer end as used in the request):

No operation: This function is blocking (unless the event has already occurred) and will return when the desired event occurs. This function is very powerful and flexible; it can be used to wait for a buffer to drain or fill up to a certain level, or just to report the status of the transfer (duplicating to some extent the information supplied by `FIONREAD` and `AIONWRITE`, although with `AIOSYNC` we can read the current size of either the FREE or the READY region of the buffer).

Send a `signal`: This function is non-blocking. It schedules a signal to be sent to the process when the buffer reaches a given mark. This provides an asynchronous notification which can be handled while a process is active, or wakes

[4] This can happen in blocking mode as well.

[5] We should have really used the name `FIONWRITE` because this function is very general and not peculiar to audio devices. E.g. it could be useful on tty devices, on network sockets and everywhere we have some amount of buffering in the hardware or the kernel.

up a process blocked on a system call.

Wakeup a selecting process: This function is non-blocking. It causes a process blocked on a `select()` call to be woken up if the desired event occurs. Note that this action is not an exact duplicate of the previous one: while a `signal` scheduled with the previous function can wake up a selecting process, there is a potential race condition in that the sequence

```
ioctl(fd, AIOSYNC, ...);
ret = select( ... );
```

might be interrupted in the middle, and the signal be delivered before the `select()` call. The problem can be solved but at the price of some obfuscation in the code. With this function, we simply request a `select()` for *exceptional conditions* on the file descriptor (specified using the fourth parameter of `select()`) to wake up when the desired event occurs. This makes us affected by timing issues because the event is possibly logged in the device driver and reported to the application as soon as `select()` is invoked.

`AIOSYNC` covers all practical needs for synchronization, and the cost of implementing the different notification methods is minimal because they share almost the same code paths.

The resolution of the `AIOSYNC` calls depends a lot on the features of the underlying hardware. On some devices, the DMA engine can be reprogrammed on the fly to generate an interrupt exactly when the desired event occurs. On other devices, this cannot be done, so if the desired event falls within the boundaries of an already started DMA transfer, there is no alternative but to periodically poll the status of the transfer. In this case, the resolution which can be achieved depends on the granularity of the system's timer, because the poll is generally done once per timer tick. Common values for the timer frequency correspond to a granularity of 1 to 10 ms, which are acceptable for the coarse synchronization of streams[6].

[6]Consider that 10 ms corresponds, at the speed of sound, to about 3 meters which is comparable to the distance between players in an orchestra; the refresh rate of video devices is in the 10 to 20 ms range, so a synchronized video output with higher resolution would be useless; moreover, non real-time operating systems would make a higher resolution useless because of the jitter in scheduling processes.

The last `ioctl()` we use to support synchronization is `AIOSTOP`. The function takes the indication of a channel and immediately suspends the transfer on that channel, flushing the content of the kernel buffer. The return value from the function is the amount of data queued in the buffer when the channel was stopped. This function allows the application to suspend a recording when it decides that no more data is required and directly supports the PAUSE function in audio players. It is responsibility of the application to reload any data that was flushed in the play buffer. There is no explicit function to start a transfer after a pause, because this action is implicit when issuing a `read()`, `write()` or `select()` call.

Function	Description
`FIONBIO`	Selects blocking or non-blocking mode of operation for the device
`FIONREAD`	Returns the amount of data which can be read without blocking.
`AIONWRITE`	Returns the amount of data which can be written without blocking.
`AIOSSIZE`	Selects character or block mode of operation for the device, setting the threshold for `select()` to return.
`AIOGSIZE`	returns the block size currently in use.
`AIOSYNC`	Schedules the requested action (return, signal, or enable select) at the occurrence of the specified event. Returns the current status of the buffer.
`AIOSTOP`	Immediately stops the transfer on the channel, and flushes the buffer. Returns the residual status of the buffer.
`read()`	Returns at most the amount of data requested. Might return a short count even in blocking mode. Also starts a paused recording.
`write()`	Writes at most the amount of data requested. Might return a short count even in blocking mode. Also starts a paused playback.
`select()`	In character mode, will return when at least one byte can be exchanged with the device. In block mode, will return when at least a full block (of the size specified with `AIOSSIZE`) can be exchanged with the device. Also start a paused recording.

Table 1: Functions supported by our audio driver for data transfer and synchronization.

For reference, Table 1 summarizes all the functions related to synchronization and data transfer supported by our driver. There is not a function to set the size of the kernel buffer in the device driver: we did not implement it on purpose, because we do not believe it to be desirable or useful. In fact, buffering within the device driver serves to avoid requiring applications to communicate with the CODEC at many small intervals. The amount of buffering required to avoid loss of data depends mainly on the speed and load of the system, and is much better decided by the operating system, rather than by the application. Also, these buffers are used by the DMA engine and reside in non-pageable memory, so it is again up to the OS to decide how many resources can be dedicated to this purpose.

It follows from the above that applications with special buffering requirements can not rely on kernel resources which might not be available to the same degree on all systems, and will need to implement buffering on their own; this also gives them greater control over buffers, and improves portability of the code. These are the same reasons which suggest not to include functions to manipulate the content of the internal buffers of the device driver.

2.4 Related Work

There is unfortunately relatively little published work on audio device drivers. Most work on multimedia devices focuses on video acquisition and rendering, which has more demanding requirements in terms of processing and data-movement overhead. Most operating systems implement a primitive interface to the audio hardware, giving only access to the basic features of the CODEC [2], and with little or no support for synchronization.

The mapping of kernel buffer in the process' memory space has gained some popularity in recent times, on the grounds that this technique can save some unnecessary copies of data [2, 3, 4]. Having the buffer mapped in memory also gives the (false) sensation that programs can gain functionality. As an example, the typical use of memory mapped buffers in audio conferencing programs is to pre-initialize the playback buffer with significant data (e.g. white noise, or silence) to minimize the effect of missing audio packets. For games, things can be arranged so that some background music is placed in the buffer and played forever without further intervention.

We believe that memory mapped access to the audio buffer is not important in a modern system, where the memory bandwidth is orders of magnitude greater than the required data rate. Provided a suitable synchronization mechanism exists, such as `AIOSYNC`, the pre-initialization of the buffer described above can be easily implemented in the application using the conventional `read()/write()` interface, also gaining in programming clarity. Additionally, for special applications such as audio conferencing, pre-filling the buffer can be efficiently done in the driver itself (as we in fact do). Finally, separate processes or threads can be used to generate background audio in a flexible way.

In many systems, access to the audio device is mediated through a library [4] which provides additional functionalities such as mixing multiple streams, playing entire files in the background, etc.. This approach is certainly advisable, although a libraries can only export and simplify the use of functionalities existing in the device driver.

Another popular approach for audio applications is to mediate access to the audio device through a server process [5, 6], similar to the X-Windows server. The very nature of audio poses some limitations to this approach. Multiplexing audio output is not as simple as for video, where multiple independent windows can be created. Thus, mechanisms are required to move the "focus" of the server from one application to the other, either manually or automatically. The second, more important, problem is related to the real-time nature of audio: mediating data transfers through an additional process, and possibly through a communication channel, can introduce further, unpredictable, delays in the communication with negative effects on some applications.

3 Video acquisition

Video acquisition devices capture data from analog video sources, encoded as PAL, NTSC, or SECAM standard color signals. The typical structure of these devices is described in Section 3.1.

The analog encoding standards for video signals are based on the sensitivity of the human eye with respect to intensity (luminance) and color (chrominance) information. Human eye receptors, nerves, are called *rods* and *cones*. The rods detect luminance or grey scale, and are used for peripheral vision. The cones detect color, details and far-away objects [7]. There are less cones than rods, and the cones have significantly less spatial resolution [8]. As a consequence, analog video standards encode separately the luminance and chrominance signal, and may utilize less bandwidth for chrominance information.

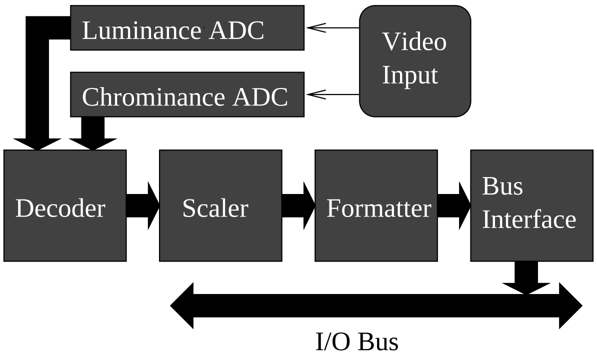

Figure 4: Typical Video Acquisition Board

3.1 Video Hardware Description

Analog PAL, NTSC, or SECAM standard color signals are decoded, scaled to the desired value, sent through a formatter and written to system memory via the I/O bus interface as shown in Figure 4. The decoder contains a luminance processor, a chroma processor, and a synchronization and clock processor. The decoder also extracts timing information from the video signals. The output from the decoder consists of samples of the luminance (Y) and chrominance (U,V) components of the signal, together with timing information. The scaler accepts the YUV data from the decoder, interpolates the samples, scales the video downward to the desired size, and filters the data in both the horizontal and vertical domains. It is extremely important that this function is done by the hardware because this operation is expensive, and scaling the image to the desired size beforehand also reduces the amount of data to be transferred to memory.

3.1.1 Video Color Space Representation

The formatter performs digital color space conversion of the YUV data into several output formats, to suit the needs of the applications in both color space representation (e.g. YUV or RGB) and data arrangement (e.g. planer or packed).

The YUV representation is oriented on the human perception of visual information, and it is the preferred format for doing compression or similar processing. The RGB representation is based on the reproduction of color information for peripherals, and it is better suited for applications like displaying on CRTs. The most common formats are RGB and YUV as described in Table 2.

RGB stands for Red, Green, and Blue which are the principle signal components of color cameras, scanners, or CRTs.

CMYK stands for Cyan, Magenta, Yellow, and Black. It describes which color component is removed from white to generate a certain color. K is defined as the minimum of CMY. The relationship of CMY to RGB us given as:

$$\begin{vmatrix} C \\ M \\ Y \end{vmatrix} = \begin{vmatrix} 1 \\ 1 \\ 1 \end{vmatrix} - \begin{vmatrix} R \\ G \\ B \end{vmatrix}$$

YUV is a color difference component representation. Y stands for luminance information and is compatible to a grey or black and white signal. U and V are the color difference signals. $U = C_B = (B - Y)/1.772$ and $V = C_R = (R - Y)/1.402$. The direct and inverse matrixes for YUV and RGB conversion [9] are:

$$\begin{vmatrix} Y \\ U \\ V \end{vmatrix} = \begin{vmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{vmatrix} \times \begin{vmatrix} R \\ G \\ B \end{vmatrix}$$

and

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} 1 & 0 & 1.402 \\ 1 & -0.344 & -0.714 \\ 1 & 1.772 & 0 \end{vmatrix} \times \begin{vmatrix} Y \\ U \\ V \end{vmatrix}$$

HSI, **HSV**, or **HSL** stands for Hue, Saturation, and Intensity, (V for Value, or L for Luminance). The Intensity, Luminance, or Value is equivalent to the Y value of YUV space. The Hue and Saturation describe the chrominance (UV) plane in polar coordinates by means of a vector, its length (S = Saturation) and its angle (H = Hue) [8].

Table 2: Color Representations

Data can be arranged by packing samples together in a pixel-by-pixel basis (*packed* format) or in a component-by-component basis (*planer* format). The choice depends on the type of processing that the application requires. Packed formats are preferred when doing color space conversion or rendering, and processing is done on all components of each pixel. Planer formats are generally used for computations that involve only one of the components at a time (e.g. DCT), in order to improve the pattern of access to memory.

Table 3 describes the color formats supported by the video acquisition driver in FreeBSD. Other formats, such as, YUV 4:1:1, YUV 4:2:0, and RGB 8 are utilized by some video capture hardware, but are not supported by our driver. Many of these formats and data types are described on the FOURCC[10] home page.

3.1.2 Frames and Fields

Analog video is interlaced and captured as two fields. Each field is denoted by either the odd or even designation. NTSC fields are separated by 1/60th second and PAL fields are separated by 1/50th of a second. To capture a full size frame, two fields (both even and odd) are interlaced and stored in memory. Thus, a NTSC input signal generates 30 frames/second and a PAL input signal generates 25 frames/second.

The time differential between even and odd fields in a frame may create some undesirable effects in the stored video image. These interlaced fields cause high frequency components to be introduced when compression algorithms are used. There are several methods of eliminating these undesired effects, one simple method, which is commonly used, is to only capture and encode the even (or odd) fields in a frame.

YUV 4:2:2 - Y is sampled at every pixel, U & V are sampled at every second pixel horizontally on each line. This format may either packed or planer. The packed pixel format has the following order: [U0 Y0 V0 Y1 U2 Y2 V2 Y3 ...] The planer format is a width×height Y plane followed by width/2×height/2 U and V planes.

YUV 12 - Y is sampled at every pixel, U & V are sampled at every second pixel horizontally on each line and every second pixel vertically. The format is planer with a width×height Y plane followed by a width/2×height/2 U and V planes.

YUV 9 - Y is sampled at every pixel, U & V are sampled at every fourth pixel horizontally and every fourth pixel vertically. The format is planer with a width×height Y plane followed by a width/4×height/4 U and V planes.

RGB 16 - There are two formats for RGB 16 data. The most commonly used format is RGB 5:5:5. Each format represents 16 bits per pixel.

RGB 5:5:5 - Each component, red, green, or blue, are represented by 5 bits giving 32 different levels of each or 32786 possible colors, the most significant bit is 0. The format is packed as follows [xRRRRRGG GGGBBBBB].

RGB 5:6:5 - The red component is represented by 5 bits, the green by 6, and the blue is represented by 5 bits. The format is packed as follows [RRRRRGGG GGGBBBBB] (this format is not supported by the driver, but included here for completeness).

RGB 32 - Each component, red, green, or blue, are represented by 8 bits giving 256 different levels of each or 16.7 million colors. The format is packed as follows: [xxxxxxxx RRRRRRRR GGGGGGGG BBBBBBBB]

Table 3: Common Digital Video Formats

3.2 Applications Interface to the Video Hardware

Application programs may request one of three video capture methods: single frame capture, asynchronous continuous capture, or synchronous pipelined capture which captures into a circular buffer with low and high water marks. Along with the type of capture, the application program must specify the size of the image to capture and what format (see Table 3) to store the captured data. Table 4 describes the standard video sizes.

Format	Full	CIF	QCIF
NTSC	640 x 480	320 x 240	160 x 120
PAL	720 x 576	384 x 288	192 x 144

Table 4: Standard Square Pixel Video Sizes [8]

3.2.1 Video Capture Modes

The FreeBSD video acquisition driver provides three modes of capture operation: the conventional Unix `read()` interface, memory mapped single capture or asynchronous continuous capture, and memory mapped synchronous multi-frame ring buffer capture.

Conventional Unix `read()` interface: This method of capture may be used with little programming cost and is used for capturing a single frame of data. In this mode, the application opens the device, sets the capture mode and size, and uses the `read()` system call to load the data into an application defined buffer. The hardware copies video data into a system memory location. When a full frame is available, the system copies this data from system memory into the user data buffer and returns control to the application program.

Memory mapped single capture or asynchronous continuous capture: In this mode, the application opens the device, sets the capture mode and size, memory maps the frame buffer into the user process space, and issues either the single-capture or the continuous-capture `ioctl()` to load the data into the memory mapped buffer. The hardware copies video data into a system memory location. In single-capture mode, the `ioctl()` call returns once the data capture is complete. In continuous-capture mode, the `ioctl()` call returns immediately and the memory buffer is updated continuously until the application requests video capture be discontinued.

Memory mapped synchronous multi-frame ring buffer capture: The application opens the device, sets the geometry, memory maps the common control structure and the data, defines the frame size, number of frames, high and low water marks and starts synchronous continuous capture mode.

Table 5 describes the memory model used by the video capture driver to synchronize the kernel with the application program. The kernel synchronizes by incrementing the count of the *number of active frames* and marking the frame as active in the *active bitmap*. If the number of active frames exceeds the *high water mark*, the driver ceases to capture data until the *number of active frames* falls below the *low water mark*. It is the applications responsibility to clear the *active bitmap* and decrement the *number of active frames* count.

Frame size	Rows × Columns × Depth.
Number of frames	Number of frames in the buffer.
Low water mark	Capture stops when the number of active frames is > than this number.
High water mark	Capture starts when number of active frames is <= this number.
Active bitmap	Bit mask of active frames, kernel sets, application clears.
Number of active frames	Count of active frames, kernel increments, application decrements.
Frame 1	
...	
Frame N	

Table 5: Memory Map of Synchronous Capture Mode

The memory mapped interface to video data was motivated by the overhead of copying the video data from system space to user space. Depending on the size and format of the requested data, there may be a substantial savings of both CPU time and memory bandwidth by avoiding this copy and directly accessing the data. More detail on this subject is discussed in the performance section (3.4).

Multiple frame buffers are useful so that the application program can utilize the current frame while the system is capturing the next frame. On slower machines several frames may be captured while the application is processing a previously acquired frame. By utilizing multiple frame buffers, the application can elect to skip intermediate frames and process the most recently acquired frame while maintaining synchronization with the video stream.

3.3 Video Synchronization

There are several synchronization tools that the video device driver supplies to application programs.

Signals: The driver will send a signal to the application program each time a frame is complete. This is useful for asynchronous notification of frame completion (in continuous mode) or frame available (in synchronous mode).

Time-stamps: The driver will append a time-stamp to each frame. The time-stamp is the time the capture of the frame was completed. Time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. This is useful for combining video streams with time-stamped audio streams.

Common Bitmap: In synchronous capture mode when a frame is captured, the frame number is marked as active in a bitmap. It is up to the application to clear the bit. This is useful for determining the most recent frame, as well as, how many frames have been captured since the last comparison of the bitmap.

3.4 Video Performance Issues

The video capture driver is able to capture frames into system memory at full NTSC or PAL speeds, provided the I/O bus can sustain the data rates required for full speed capture. These rates are shown in Table 6. With the introduction of the Intel Triton chip set (430TX), full speed, full frame video acquisition was possible on the PC.

Analog Format	Speed (f/s)	Size (pix/f)	Rate (RGB 32) (MBytes/sec)
NTSC	30	640x480	36.864 MB/s
PAL	25	720x576	41.472 MB/s

Table 6: Data Rates for Video Acquisition

One possible application for the video acquisition device would be to have the acquisition hardware place the digitized video data into memory and then copy this data to the video display device (e.g. a TV viewer). This application would not have to compress the data, but would have to move the data from memory to the frame buffer on the video display device.

Memory bandwidth on a common PC can range from 10 MB/s to 100 MB/s [14]. More recent PC interface chipsets have memory bandwidths on the order of 200 MB/s [15] [7]. On some machines, just the simple movement of the video data, such as a TV viewer, may consume the entire memory bandwidth. Other applications, that require compression and decompression of the video data, will take additional memory bandwidth and CPU power.

Because of the bandwidth required for full size, full motion video (~40 MBytes/second), limited memory bandwidths, and available CPU power, it is difficult to achieve full motion compression of video in real-time without hardware support.

4 Availability and Implementation Status

The audio driver presented in this paper is now a standard component of the FreeBSD operating system, and it is in widespread use, with most legacy applications already working with it either unmodified, or with minor modifications to the audio module. Because the new driver has better support for full duplex operation than the previously existing Voxware driver, audio conferencing tools like vat [16] and rat [17, 18] can now be used on a wide range of hardware. Compared to the previously existing driver, the new one has a much simpler configuration, because a single `device pcm ...` entry in the kernel configuration file brings in support for a number of different cards. The task of identifying the correct card type, and doing specific resource configuration, is now left to the driver rather than to the user at system configuration time.

Newer and beta releases of the driver, to support more cards and fix bugs, are available from the author's home page [19]. At the time of this writing, all of the supported cards use the services of the ISA DMA controller to support DMA operations. As a consequence, most of the functionalities described in this paper could be supported by using some simple code to fetch the transfer status from the ISA DMA controller. In order to obtain the asynchronous notifications needed to wake up sleeping processes, two approaches have been followed. If the audio device supports interrupting a DMA operation on the fly, then the device is reprogrammed to generate an interrupt when the desired event occurs. When this is not possible, a periodic handler is scheduled to process the event within one timer tick from its occurrence. The overhead for the periodic handler is

[7] At the time of writing of this paper, a newly introduced Intel chip set (440BX) has been reported to have a memory bandwidth of 300 MB/s and CPUs are running at 400 Mhz. Even at these rates it will be difficult to encode, decode, and display video streams on a single PC without hardware assistance.

very small, and the resolution is 10 ms with the default timer frequency (100 Hz).

The video acquisition driver is part of the standard distributions of FreeBSD[8]. Separate `device...` entries are required for the two most popular PCI-based frame grabbers: the Matrox `meteor` which based on the Philips chipsets, and the `bktr` which based on the Brooktree 848/849 chipsets.

Many applications can utilize the digitized video provided by the video acquisition driver in FreeBSD. The motivation for writing the driver was to work with network streaming video conferencing programs such as nv [12], and vic [13]. Both of these programs use the video acquisition driver in continuous capture mode and incorporate software compression algorithms. Because of the performance of PC memory architecture (see section 3.4) and the amount of data (see Table 6), full frame, full rate data compression is difficult to achieve on a current day PC.

Fxtv [20] is a recent application that uses the video acquisition device to write digitized video data directly to the linear frame buffer of the video display device. Both the `meteor` and the `bktr` drivers support direct PCI to PCI bus transfers. Because this application doesn't require real-time compression, and it utilizes the PCI to PCI transfer capabilities, it can maintain full speed, full motion video on the display device without affecting available memory bandwidth or processor power.

The video driver interface does not currently implement the `select` system call. For consistency and synchronization purposes, this call should be added to the video driver.

5 Conclusions

We have described software interfaces for audio and video acquisition devices to improve support for multimedia applications. In defining these interfaces we have tried to pursue the following goals:

- look at the requirements of applications, rather than trying to extend an existing software interface;
- only specify the external interface of the device driver; do not rely or make assumptions about the internal structure of the driver or of the hardware; Do not export information which could lead to non-portable code to be written;
- keep the number of functions small;
- avoid duplication of functionalities in the interface, so that there is no doubt on what is the preferred method to achieve a given result.

We believe we have achieved the above goals, because our interfaces are small, powerful and simple to use, and resulted in a very compact implementation.

Hardware compression and decompression solutions will have to be implemented in order to allow real-time storage and retrieval of video data. "Currently, there are too many compression algorithms and standards and too few low-cost boards that implement the major standards [21]." One hardware solution which looks promising is the MPACT [22] chipset.

The video interface functions well and has been extended to work with the Brooktree 848 [11] video capture chip; however, an extensible generic multimedia kernel level interface with integrated and synchronized audio, video, compression, and decompression will be required for advanced applications. Some thought has been given to this interface and it still under discussion on the `freebsd-multimedia@freebsd.org` mailing list.

Acknowledgments

This work has been partly supported by the Ministero dell' Universita' e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the Project "Design Methodologies and Tools of High Performance Systems for Distributed Applications". It has also been partially supported by the Electrical Engineering and Computer Science Department at the University of Wisconsin-Milwaukee.

The authors wish to thank Mark Tinguely, co-author of the meteor device driver, and Amancio Hasty, who wrote the Bt848 device driver and has provided invaluable support to many of the multimedia drivers and applications. There are many

[8] The meteor driver has also been ported to the Linux operating system.

users on the freebsd-multimedia@freebsd.org mailing list who have also provided invaluable support.

References

[1] "The FreeBSD operating system home page," http://www.freebsd.org/.

[2] P. Bahl, "The J300 Family of Video and Audio Adapters: Software Architecture", Digital Technical Journal vo.7 n.4, 1995, pp.34-51

[3] The Open Sound System (OSS) Web page, http://www.4front-tech.com/.

[4] Microsoft Corp., Documentation on the DirectSound SDK, available at http://www.microsoft.com/DirectX/.

[5] T.M. Levergood, A.C. Payne et al., "AudioFile: Network-Transparent System for Distributed Audio Applications", USENIX Summer Conference 1993, June 1993.

[6] J. Fulton, G. Renda, "The Network Audio System", 8th Annual X Technical Conference, in "The X Resource, Issue Nine, January 1994".

[7] "Flight Training Handbook," U.S. Department of Transportation, Federal Aviation Administration, pp 194-195, AC 61-21A, 1980.

[8] "Color space, digital coding, and sampling schemes for video signals," Desktop Video Data Handbook, Philips Semiconductors, Data Handbook IC22, pp 76, 1995.

[9] CCIR (Consultative Committee on International Radio) "Recommendation 601-2: Encoding Parameters of Digital Television for Studios," 1982-1986-1990.

[10] "The FOURCC (Four Character Code) home page," http://www.webartz.com/fourcc/.

[11] "Bt848: Single-Chip Video Capture for PCI," Data Sheet, Brooktree Corporation, August 1996.

[12] Ron Frederick, "Experiences with real-time software video compression," in Sixth International Workshop on Packet Video, July 1994.

[13] Steve McCanne and Van Jacobson, "vic: A flexible framework for packet video," in Proc. of ACM Multimedia '95, Nov. 1995.

[14] Larry McVoy and Carl Staelin, "lmbench: Portable tools for performance analysis", Usenix proceedings, January 1996.

[15] John D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," IEEE Technical Conference on Computer Architecture newsletter, December 1995.

[16] V.Jacobson, S.McCanne: "The LBL audio tool vat", Manual page, http://www-nrg.ee.lbl.gov/vat/.

[17] V.Hardman, M.A.Sasse, M.Handley, A.Watson: "Reliable audio for use over the Internet", INET'95 conference.

[18] V.Hardman, I.Kouvelas, M.A.Sasse, A.Watson: "A packet loss Robust Audio Tool for use over the Mbone", Research Note RN/96/8, Dept. of Computer Science, University College London, 1996.

[19] "Luigi Rizzo's FreeBSD home page," http://www.iet.unipi.it/~luigi/FreeBSD.html.

[20] "Randall Hopper's FreeBSD X TV home page," http://multiverse.com/~rhh/fxtv/.

[21] L.A. Rowe, "Video Compression for Desktop Applications",... Informationstechnik und Technische Informatik, Vol 37, No 4, pp 7-10, August 1995.

[22] "The MPACT home page," Chromatic Research, http://www.mpact.com/.