The FreeBSD Audio Driver

Luigi Rizzo

Dip. di Ingegneria dell'Informazione
Università di Pisa
via Diotisalvi 2 – 56126 Pisa (Italy)
email: `l.rizzo@iet.unipi.it`

Abstract. We recently developed an audio driver in the FreeBSD operating system. In this work, we decided to consider compatibility with existing software interfaces only as a secondary issue, to be implemented at a later time and only for those applications which could not be adapted to the new software interface. This turned out to be a significant advantage, since it let us design the driver (and particularly, its software interface) looking at the real needs of applications, rather than duplicating existing, old interfaces, and having applications adapt (in many cases suboptimally) to what the driver could offer.
The main results of our work is the definition of a software interface for audio devices which is well suited to multimedia applications. The new interface is small, simple but powerful, and allowed several simplifications, and significant performance enhancements, in the applications. In this paper we motivate our design choices, illustrate our interface, and discuss implementation issues both for the device driver and applications. The software described in this paper, and appropriate application routines, are available from the author.
Keywords: Multimedia, audio conferencing, audio devices, operating systems.

1 Introduction

Networked multimedia applications have become very popular in recent years [4, 6, 12, 13], due to a number of enabling factors such as the availability of high performance computing hardware with multimedia capabilities, which have made it possible to run in software powerful compression/decompression algorithms, and advances in network and modem technology, which have made network connectivity very widespread and with suitable data rates.

Among multimedia applications, the most compelling ones are those requiring (soft) real-time features, synchronization among streams of different types, or full duplex operations. Audio/video conferencing systems are typical examples where these demanding requirements appear.

Operating system support for multimedia devices, however, predates these applications. Audio CODECS and video grabbers are conceptually simple devices, and in principle they can be easily integrated into an operating system (e.g. Unix) using standard primitives such as **`read()`** and **`write()`**, plus a handful of **`ioctl()`** calls to access special device features. Such a primitive audio device driver[1] is what is generally available in Unix and other systems, under the assumption that additional functionalities can be supplied by library routines, or by passing audio data through a server process which acts similarly to the X Window server (see Fig. 2).

The reason why the above reasoning fails is that the server will ultimately need the services of the device driver as well. As more and more functionalities are put in the hardware (e.g. by offloading operations to a DSP in the audio card) the software interface of the device driver will need to be extended to effectively support these functionalities.

But what are really the needs of applications which require support within the device driver ? Buffering is often almost all what is needed if one of the communicating parties has no strong timing requirements (e.g. it is a disk), and can adapt to the average data rate that the CODEC imposes. However, full duplex applications, or other applications with real-time constraints (e.g. when audio must be synchronized with visualization data), have some timing requirements, since they need to act upon specific events (e.g. when a given sample is acquired or played out). This in turn requires the software interface to the device to support the exchange of synchronization information. When these are not available (e.g. because they cannot be easily accomodated in some preexisting device driver structure, or because they would break backward compatibility), generally the only alternative is to enable non-blocking operation on the audio device, and implement synchronization-related functionalities in other ways (e.g. by means of timers). But this is an expensive solution, since it makes applications busy-wait for events instead of using the synchronization mechanism which are available in the kernel. Yet this is a common approach, which is bearable only because today's workstations are sufficiently overpowered to tolerate even highly inefficient operating systems and applications.

Recently, we were involved in the development of a new audio driver for the FreeBSD operating system. FreeBSD did have a freely available device driver for sound cards, but it lacked support for newer cards, lacked functionalities (e.g. full duplex support) and had some limitations which applications had to circumvent at the price of performance, efficiency or program clarity. Our goal was then to produce a new driver to overcome all these limitations, possibly redefining the software interface of the driver. On the other hand, the existing software interface [10] was sufficiently popular to require support in the new driver.

This gave us two alternatives: either build a driver which was compatible with the existing one, adding new features on top of this; or start the design without the constraint of backward compatibility, and add a compatibility layer at a later time, only for those applications for which source code was not available and adaptation to the new interface was not possible. Clearly the second approach was more demanding, in that it required a study of existing applications to determine which features were needed and how they were used, and some

[1] In this paper we will not consider specialized hardware which exists for generating music, e.g. FM or wavetable synthesizers. These devices have a completely different internal structure and software interface.

porting effort to adapt applications to the new interface. However, this approach proved to be very fruitful. In fact, the analysis of applications let us understand what the driver *should really* do, rather than what the driver *used* to do. This analysis, not without surprise, showed that many applications were adapted to the existing audio driver in a quick and dirty way, by making inconsistent use of the available functions, calling functions with the same or similar semantics in mixed or redundant ways, and very often using features (such as non-blocking I/O) which were not necessary or highly expensive. As a result of this analysis, we had a chance of identifying which of the existing functionalities were really needed, which ones were merely duplicates, which ones were probably useless, and which ones were missing. This served us as a guide to what we should really implement in our driver and, at the same time, to how the applications we studied should make use of the audio functions.

Basing on this work, we have defined what we believe to be a simple yet powerful set of functions to access audio devices. These functions can be easily and efficiently implemented in device drivers, let applications make full use of the available synchronization mechanisms, and give very good support for highly demanding multimedia applications. In this paper we provide a detailed description of the functionality and the semantics of our primitives, motivating their existence and showing their usefulness. The software interface described in this paper has been implemented and application have been modified to make use of it, often with large improvements on the efficiency of the applications, and also with some simplifications in the code.

The paper is structured as follows. We start with a basic overview of the hardware which implements audio devices. In Section 3 we describe the methods to access the device to transfer data, and motivate our choices in relation with some existing applications. In Section 4 we discuss the synchronization requirements between applications and the driver, and show how a small set of calls can be used to implement them efficiently. Related work is presented in Section 5. Finally, in Sections 6 we discuss some implementation issues and show how the new features were used to improve the performance of existing applications.

2 Audio hardware

Figure 1 shows the hardware/software structure of a typical audio device. We have two logically independent subsystems[2], one for audio capture and one for audio playout.

The capture section contains a *mixer* to select the input source which is routed to the Analog to Digital Converter (ADC). The ADC output is stored into a circular buffer, generally by means of some DMA engine. The hardware dictates the native resolution (e.g. 8..16 bits), data formats (e.g. linear or companded using μ-law), channels (mono/stereo) and sampling rate, whereas the software

[2] except for cases where hardware limitations make the subsystem share resources thus preventing full duplex operation.

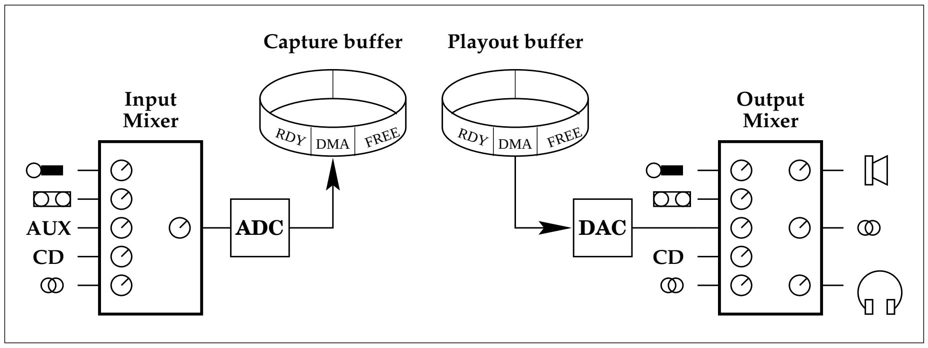

Fig. 1. A model of audio devices. On the left is the capture section, on the right the playout section.

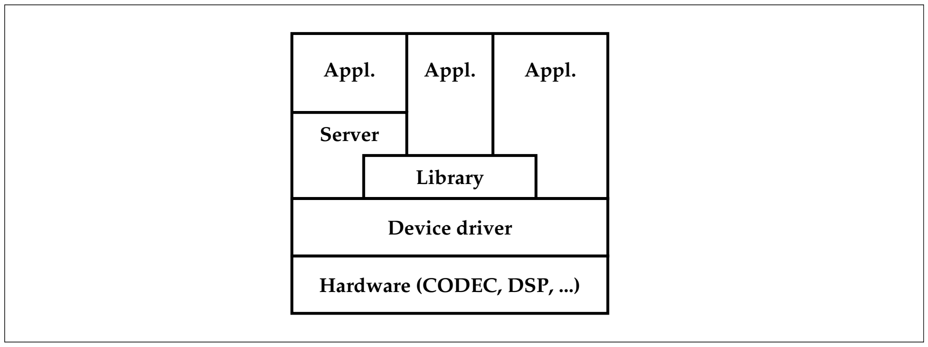

Fig. 2. Possible structure of applications using the audio device. Access can be direct, through a library, or through a server.

implements the circular buffer and may additionally implement some data format conversion before passing data to the application program.

The playout section uses a circular buffer to hold data which are sent to the Digital to Analog Converter (DAC), generally using a DMA engine. The output of the DAC is routed through a mixer, together with other sources, to the appropriate output. Again, 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 called *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 times, the audio device uses a DMA channel which is present on the 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 the PCI bus generally operate in this way). As an alternative, the audio device could have some internal memory where data are 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 onboard 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 good idea since some algorithms (e.g. GSM or MPEG encoding/decoding) are expensive and it is much cheaper to run them on a DSP (which has become a commodity device since 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 on the various channels on the output section. Newer devices, finally, have full-featured mixers on the input path as well, together with a larger selection of sources and the ability of performing miscellaneous functions such as swapping left and right channels, muting sources, etc.

3 Software interface to the audio device

In this Section we start the presentation of our software interface to audio devices. We focus our attention on the "device driver" layer in Figure 2, and particularly to the transfer of data between the audio device and the application program. Synchronization primitives will be discussed in the next Section.

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 "feature"[3]), and to set the desired data formats. These operations are usually done by means of some `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 since this is generally a one-time operation.

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

[3] There are significant differences among audio cards. As an example, some CODECS can only work in full duplex under some constraints, e.g. using 8-bit format in one direction and 16-bit in the other one. Other CODECS have bugs which 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 that the driver provides a unique identifier for the actual hardware and applications (or libraries) can map these identifiers to a list of features and adapt to them.

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 since 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.

We will not discuss the three items above (selecting data format, programming onboard devices, and controlling the mixer) in further detail in this paper.

3.1 Character versus block mode

Traditionally, audio devices have been used as character devices, with the granularity of access supported by the driver being the single byte. Applications however often transfer data in blocks of fixed size. This is not only a matter of convenience, it is also a need dictated by the use of compression algorithms which operate on blocks of samples with a given size. This pattern of access has implications on the type of operating system support required by such applications.

Multimedia applications usually have to deal with multiple events (e.g. audio input, data coming from the network, GUI events, timers, etc.) and they cannot block on a single source waiting for data to be ready. The usual solution to this problem is to use a blocking primitive (such as the Unix `select()`) which allows the process to block specifying a set of events which the process is waiting for. If the audio device is treated as a character device, a `select()` on the device might return as soon as a single byte can be transferred – at most 125 μs with the typical sample rate of audioconferencing tool, or even less for high quality audio. What we would really want, instead, is to specify to the driver the granularity of the `select()` operation, so that it will not wake up until the desired amount of data is ready[12]. Some software interfaces, e.g. OSS (formerly Voxware) [10], allow this but only indirectly, since they permit to choose among a small number of block sizes for DMA operations.

To fulfill this goal, our 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` can modify the requested value if it is not an acceptable one (e.g. the requested block size exceeds the size of the internal buffer) and returns the block size in use. A block size of 0 or 1 will bring the device driver back into character mode.

We want to point out that the `AIOSSIZE` function only specifies the behaviour of the `select()` call. In our driver, 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, by no means the user should make further assumptions on the behaviour 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 granularity of DMA transfers equals the value specified with `AIOSSIZE`). Such assumptions might make programs not check error conditions (e.g. `read()` or `write()` returning with a short count) which might indeed occur depending on the internal implementation of the driver or the features of the hardware.

3.2 Support for non-blocking I/O

Traditionally, device drivers provide support for non blocking I/O. This operating mode can be selected at device open time, or 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 a dual of `FIONREAD` for write operations. In our driver we have implemented such a function, called `AIONWRITE`[5], which returns the free space in the playout buffer. A write of this many bytes will not block, even if blocking mode is selected. In our implementation, both `FIONREAD` and `AIONWRITE` track closely the status of a DMA transfer.

4 Synchronization

In this Section we discuss in more detail those which we consider useful features to support synchronization of audio with other activities.

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 should be synchronized with the audio streams. As an example, a player program wants to avoid that the playout buffer becomes empty in the middle of operation; or, a telephone-like application might need to check that the buffers do not become too full, in order to control the end-to-end delay.

When data transfer occurs at a constant nominal rate, in principle this information can be derived by using a real time clock. However, this method can be imprecise because there might be deviations between the nominal and the actual sample rate, drifts in the two clocks, or buffer overflows/underflows which cause samples to be missed. Furthermore, variable-rate compression schemes, or

[4] this could happen anyways even in blocking mode.

[5] we should have really used the name `FIONWRITE` since 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.

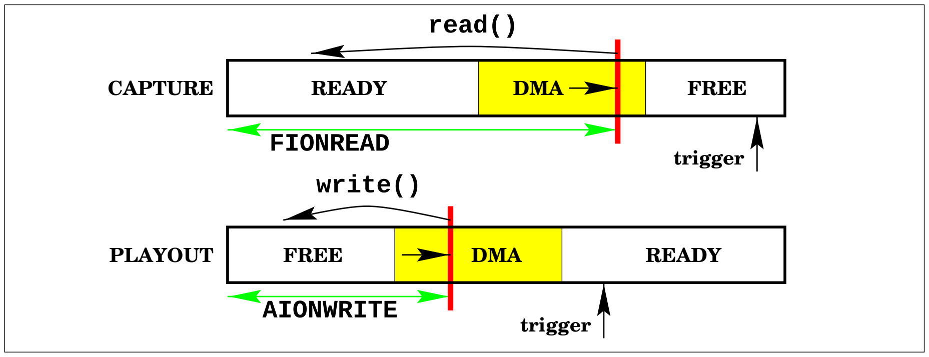

Fig. 3. A view of the capture and playout buffers, with the indication of events which trigger the actions requested with `AIOSYNC`. The thick vertical bar indicates the current position of the DMA pointer. Also indicated are the effect of `read()` and `write()`, and the values returned by `FIONREAD`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

dynamically-changing buffer sizes, might render the use of timers for determining queue occupation completely useless. As an example, it is easy to conceive a device where some functions (e.g. DFT, compression, rate conversion, filtering) are performed in the DSP associated to the CODEC, and the application transfers audio data in a transformed domain or with a variable-rate format. In these cases, deriving timing information is not straightforward.

`FIONREAD` and `AIONWRITE` only provide limited information on the status of buffers, and are of use mainly when the application wants 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 capture and playout buffers. The boundary between the FREE and READY regions, which we call the *current DMA pointer*, moves with time (in opposite directions for the capture and playout channel). Applications can, in general, be interested to know when the current DMA pointer reaches a given position, relative to either the beginning or the end of the buffer, and have the need to be notified when the event occurs, and to know 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 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 it is waiting for exceptional conditions on the audio file descriptor and 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 not affected by timing issues since 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 since 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, since the poll is generally done once per timer tick. Common values for the timer frequency correspond to a granularity of 1..10 ms, which are acceptable for the coarse synchronization of streams (consider that 10 ms correspond, at the speed of sound, to about 3 meters, comparable to the distance between players in an orchestra; the refresh rate of most video devices is in the 10-20 ms range, so a synchronized video output with higher resolution would be useless; moreover, non real-time operating system would probably make a higher resolution not very useful 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 capture as soon as it decides that no more data are needed, and directly supports the PAUSE function in audio players. It is responsibility of the application to reload any data that was not used in the play buffer. There is no `ioctl` to start a transfer, since 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. A count of 0 or 1 means the usual character mode, a count > 1 makes select return only when the specified amount of data is available. `AIOGSIZE` returns the 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. In non-blocking mode, it will return immediately with the data already available in the buffer. Also start a paused capture.
`write()`	Writes at most the amount of data requested. Might return a short count even in blocking mode. In non-blocking mode, it will return immediately after filling up some amount of the free space in the kernel buffer. Also start a paused playout.
`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 transfer.

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. It might surprise that there is no function to set the size of the buffer in the device driver. We do not believe it to be useful, for the following reasons:

– the purpose of the buffer in the device driver is to avoid that applications have to be scheduled too often to communicate with the CODEC. A busy system, or a system where the scheduling clock runs at low frequency, needs larger buffers than one where timeslices are very short. But these parameters

are not readily available to applications, so it is the operating system which should decide how big the internal buffers should be;
- being handled by the DMA engine, these buffers must reside in non-pageable memory, and take resources permanently on the system. As a consequence the decision of how much buffer space to use *in the kernel* is not something that a single application can take, but should be taken by the operating system itself depending on the actual resources (total memory, buffers used for other devices, etc.) which are available;
- applications will need to have their own buffering in almost all cases, and they have much greater control on user-space buffers than on kernel-space buffers. In order to write more portable and reliable software, it is much better to force applications to provide their own buffering scheme, suited to the actual needs, than to rely on resources which might not be available to the same degree on all systems, and which impose stronger limitations on their use (e.g. because it is much harder to remove data from the kernel play buffer than from the user-space play buffer).

As a consequence, we left out such a call on purpose. We believe that portability and clarity of programs can improve if they do not depend on being able to set the buffer size in the kernel, and instead provide any required buffering within themselves. These are the same reasons which suggested not to include functions to manipulate the content of the internal buffers of the device driver.

5 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 [1], 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 [1, 10]. Having the buffer mapped in memory also gives the (false) sensation that programs can gain functionality. As an example, the typical use of mmapped buffers in audio conferencing programs is to pre-initialize the playout 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.

It is evident that the above are non-problems in a modern system when dealing with audio data. Audio samples have a rate of 192 KB/s at most, whereas the memory bandwidth of modern systems is 2-3 orders of magnitude higher, so the copy overhead is minuscule (for video it would be all a different story). 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 in our driver). Finally, separate processes or threads can be used to generate background audio even in a more flexible way.

In many systems, access to the audio device is mediated through a library [9] 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 [8, 3], similar to the XWindow server. The very nature of audio poses however 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.

6 Discussion and performance

Most of the primitives described in the previous Sections have been implemented in our audio driver [11] for the FreeBSD operating system [2]. The driver supports a variety of audio cards, with different features and limitations. 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. In case 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 very small, and the resolution is 10 ms with the default timer frequency (100 Hz).

6.1 Applications

We have used our new driver in a number of audio applications. In many cases, source code was available and we could update the applications to make use of the new interface, or simplify the code because the new functionalities were simpler to use.

The `AIOSIZE` call has been used to improve the behaviour of audioconferencing programs such as `vat`. `vat` transfers data in blocks of 160 bytes, corresponding to 20 ms of audio sampled at 8 KHz. The choice of the frame size is related to the compression algorithms (GSM, LPC) used in the program. The main body of the program calls `select()` on a set of file descriptors, which include the network socket, the X server, and the capture audio device. When operating in character mode, the `select()` would return after 125 μs, causing the process to consume a huge amount of CPU time just to read one character at a time and loop waiting for a full frame to be available. With block mode operation, we reduced the CPU occupation from about 50% to a mere 3% when not using compression. The advantage here (and in other similar cases) comes mostly from the availability of new synchronization mechanisms.

The `AIOSYNC` call, has been used in `vat` to control the length of the READY region of the playout buffer (the same goal can be achieved with `AIONWRITE` if the buffer size is known). `vat`, `rat` and other audio tools try to control the length of the playout buffer basing on the assumption that the capture and playout sections work at exactly the same sample frequency [5, 6, 7]. Under this assumption, the length of the playout buffer can be kept constant if as many bytes are written as they have been read from the capture section. There are situations where the assumption does not hold, e.g:

- when two physically distinct devices are used, sample clocks might be slightly different or drift over time;
- if DMA transfers are disabled while restarting a DMA operation, the average sample rate on each channel depends on how samples are lost/delayed while the DMA engine is restarted.

We have also encountered buggy CODECS which miss samples during regular operations. In all these cases, using the amount of data read as a measure of the speed of the write channel does not help, and in the long term it will cause underruns to occur, or data to accumulate in the output buffer, with consequent clicks or delays in full duplex operation. `AIOSYNC` provides a way to detect the occurrence of such events, and compensate them. In fact, before issuing a `write()` operation, the queue length can be read. If the value, in the long term, differs significantly from what is expected, the size of the write is slightly adjusted in the opposite direction to compensate the difference before the error becomes too large.

The asynchronous notification mechanisms provided by `AIOSYNC` have many other applications. Consider, as an example, a player process being run on a workstation in the background, e.g. doing MPEG decoding in software. During normal activities, the process has plenty of CPU available, and can keep the playout buffer almost full at all times. When the machine becomes loaded (e.g. during a compilation), the buffers might drain with the risk of having pauses in the audio output. The player process could then program a `signal` to be sent when the buffer reaches some low water mark, and switch to some less expensive operating mode (e.g. mono instead of stereo) upon reception of the signal.

As another application of `AIOSYNC`, a multimedia application might want to launch a free-running animation while some audio description is played on the audio device. The application could then schedule a `signal` to be sent when the buffer becomes almost empty, and start running the animation without worrying about the audio. This is useful since the animation code might use blocking primitives (e.g. while reading data from the disk, or sleeping between subsequent frames), and taking care of the audio would require heavy program restructuring or a separate thread. In our case, instead, the `signal` handler will take care of sending more data to the audio device, or forcing the termination of the animation when the appropriate conditions occur.

Other applications of the `AIOSYNC` calls include games and all those programs where visualizations or other actions (e.g. controlling an external instrument) should be done synchronously with audio events.

7 Conclusions

We have described a software interface for audio devices to improve support for multimedia applications. In defining this software interface we have tried to pursue the following goals:

– look at what could be the real needs of applications, rather than try to extend some existing software interface;
– only specify the external interface of the device driver, do not rely or make assumption on 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, since our interface is small, powerful and simple to use, and resulted in a very compact implementation of device drivers. Our initial experience in porting applications to the new interface has been highly positive, since in many cases software modules which access the audio devices could be largely simplified and in some cases improved by making use of the functions provided by our interface.

Acknowledgements

The work described in this paper has been supported in part by the Commission of European Communities, Esprit Project LTR 20422 – "Moby Dick, The Mobile Digital Companion (MOBYDICK)", and in part by the Ministero dell'Università e della Ricerca Scientifica e Tecnologica of Italy.

References

1. P. Bahl, “The J300 Family of Video and Audio Adapters: Software Architecture”, Digital Technical Journal vo.7 n.4, 1995, pp.34-51
2. The FreeBSD operating system Web page, `http://www.freebsd.org/`
3. J. Fulton, G. Renda, “The Network Audio System”, 8th Annual X Technical Conference, in “The X Resource, Issue Nine, January 1994”.
4. V.Hardman, M.A.Sasse, M.Handley, A.Watson: “Reliable audio for use over the Internet”, INET’95 conference.
5. 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.
6. V.Jacobson, S.McCanne: “The LBL audio tool vat”, Manual page (`ftp://ftp.ee.lbl.gov/conferencing/vat/`)
7. I.Kouvelas, V.Hardman: “Overcoming Workstation Scheduling Problems in a Real-Time Audio Tool”, Proc. of Usenix 1996.
8. T.M. Levergood, A.C. Payne et al., “AudioFile: Network-Transparent System for Distributed Audio Applications”, USENIX Summer Conference 1993, June 1993.
9. Microsoft Corp., Documentation on the DirectSound SDK, available at `http://www.microsoft.com/DirectX/`
10. The Open Sound System (OSS) Web page, `http://www.4front-tech.com/`
11. L.Rizzo, Sources for the new FreeBSD audio driver, available from `http://www.iet.unipi.it/~luigi/FreeBSD.html`
12. H.Schulzrinne: “Voice communication across the Internet: A Network Voice Terminal”, Technical Report TR 92-50, Dept. of Computer Science, University of Massachusets, Amherst, July 1992.
13. T.Turletti: “The inria videoconferencing system (ivs)”, ConneXions – The Interoperability Report, 8(10):20-24, October 1994.