Paramvir Bahl

The J300 Family of Video and Audio Adapters: Software Architecture

The J300 family of video and audio products is a feature-rich set of multimedia hardware adapters developed by Digital for its Alpha workstations. This paper describes the design and implementation of the J300 software architecture, focusing on the Sound & Motion J300 product. The software approach taken was to consider the hardware as two separate devices: the J300 audio subsystem and the J300 video subsystem. Libraries corresponding to the two subsystems provide application programming interfaces that offer flexible control of the hardware while supporting a client-server model for multimedia applications. The design places special emphasis on performance by favoring an asynchronous I/O programming model implemented through an innovative use of queues. The kernel-mode device driver is portable across devices because it requires minimal knowledge of the hardware. The overall design aims at easing application programming while extracting real-time performance from a non-real-time operating system. The software architecture has been successfully implemented over multiple platforms, including those based on the OpenVMS, Microsoft Windows NT, and Digital UNIX operating systems, and is the foundation on which software for Digital's current video capture, compression, and rendering hardware adapters exists.

Background

In January 1991, an advanced development project called Jvideo was jointly initiated by engineering and research organizations across Digital. Prior to this endeavor, these organizations had proposed and carried out several disjoint research projects pertaining to video compression and video rendering. The International Organization for Standardization (ISO) Joint Photographic Experts Group (JPEG) was approaching standardization of a continuous-tone, still-image compression method, and the ISO Motion Picture Experts Group's MPEG-1 effort was well on its way to defining an international standard for video compression.[1,2,3] Silicon for performing JPEG compression and decompression at real-time rates was just becoming available. It was a recognized and accepted fact that the union of audio, video, and computer systems was inevitable.

The goal of the Jvideo project was to pool the various resources within Digital to design and develop a hardware and software multimedia adapter for Digital's workstations. Jvideo would allow researchers to study the impact of video on the desktop. Huge amounts of video data, even after being compressed, stress every underlying component including networks, storage, system hardware, system software, and application software. The intent was that hands-on experience with Jvideo, while providing valuable insight toward effective management of video on the desktop, would influence and potentially improve the design of hardware and software for future computer systems.

Jvideo was a three-board, single-slot TURBOchannel adapter capable of supporting JPEG compression and decompression, video scaling, video rendering, and audio compression and decompression—all at real-time rates. Two JPEG codec chips provided simultaneous compression and decompression of video streams. A custom application-specific integrated circuit (ASIC) incorporated the bus interface with a direct memory access (DMA) controller, filtering, scaling, and Digital's proprietary video rendering logic. Jvideo's software consisted of a device driver, an audio/video library, and applications. The underlying

ULTRIX operating system (Digital's native implementation of the UNIX operating system) ran on workstations built around MIPS R3000 and R4000 processors. Application flow control was synchronous. The library maintained minimal state information, and only one process could access the device at any one time. Hardware operations were programmed directly from user space.

The Jvideo project succeeded in its objectives. Research institutes both internal and external to Digital embraced Jvideo for studying compressed video as "just another data type." While some research institutes used Jvideo for designing network protocols to allow the establishment of real-time channels over local area networks (LANs) and wide area networks (WANs), others used it to study video as a mechanism to increase user productivity.[4–8] Jvideo validated the various design decisions that were different from the trend in industry.[9] It proved that digital video could be successfully managed in a distributed environment.

The success of Jvideo, the demand for video on the desktop, and the nonavailability of silicon for MPEG compression and decompression influenced Digital's decision to build and market a low-cost multimedia adapter similar in functionality to Jvideo. The Sound & Motion J300 product, referred to in this paper as simply the J300, is a direct descendent of the Jvideo advanced development project. The J300 is a two-board, single-slot TURBOchannel option that supports all the features provided by Jvideo and more. Figure 1 presents the J300 hardware functional diagram, and Table 1 contains a list of the features offered by the J300 product. Details and analysis of the J300 hardware can be found in "The J300 Family of Video and Audio Adapters: Architecture and Hardware Design," a companion paper in this issue of the *Journal*.[9]

The latest in this series of video/audio adapters are the single-board, single-slot peripheral component interconnect (PCI)–based FullVideo Supreme and FullVideo Supreme JPEG products. These products are direct descendants of the J300 and are supported under the Digital UNIX, Microsoft Windows NT, and OpenVMS operating systems. FullVideo Supreme is a video-capture, video-render, and video-out-only option; whereas, FullVideo Supreme JPEG also includes video compression and decompression. In keeping with the trend in industry and to make the price attractive, Digital left out audio support when designing these two adapters.

All the adapters discussed are collectively called the J300 family of video and audio adapters. The software architecture for these options has evolved over years from being symmetric in Jvideo to having completely asymmetric flow control in the J300 and FullVideo Supreme adapters. This paper describes the design and implementation of the software architecture for the J300 family of multimedia devices.

Software Architecture: Goals and Design

The software design team had two primary objectives. The first and most immediate objective was to write software suitable for controlling the J300 hardware. This software had to provide applications with an application programming interface (API) that would hide device-specific programming while exposing all hardware capabilities in an intuitive manner. The software had to be robust and fast with minimal overhead.

A second, longer-term objective was to design a software architecture that could be used for successors to the J300. The goal was to define generic abstractions that would apply to future, similar multimedia devices. Furthermore, the implementation had to allow porting to other devices with relatively minimal effort.

When the project began, no mainstream multimedia devices were available on the market, and experience with video on the desktop was limited. Specifically, the leading multimedia APIs were still in their infancy, focusing attention on control of video devices like videocassette recorders (VCRs), laser disc players, and cameras. Control of compressed digital video on a workstation had not been considered in any serious manner.

The core members of the J300 design team had worked on the Jvideo project. Experiences gained from that project helped in designing an API with the following attributes:

- Separate libraries for the video and audio subsystems
- Functional-level as opposed to component-level control of the device
- Flexibility in algorithmic and hardware tuning
- Provision for both synchronous and asynchronous flow control
- Support for a client-server model of multimedia computing
- Support for doing audio-video synchronization at higher layers

In addition, the architecture was designed to be independent of the underlying operating system and hardware platform. It included a clean separation between device-independent and device-dependent portions, and, most important, it left device programming in the user space. This last feature made the debugging process tractable and was the key reason behind the design of a generic, portable kernel-mode multimedia device driver.

As shown in the sections that follow, the software design decisions were influenced greatly by the desire to obtain good performance. The goal of extracting real-time performance from a non-real-time operating system was challenging. Toward this end, designers placed special emphasis on providing an asynchronous model for software flow control, on designing a fast

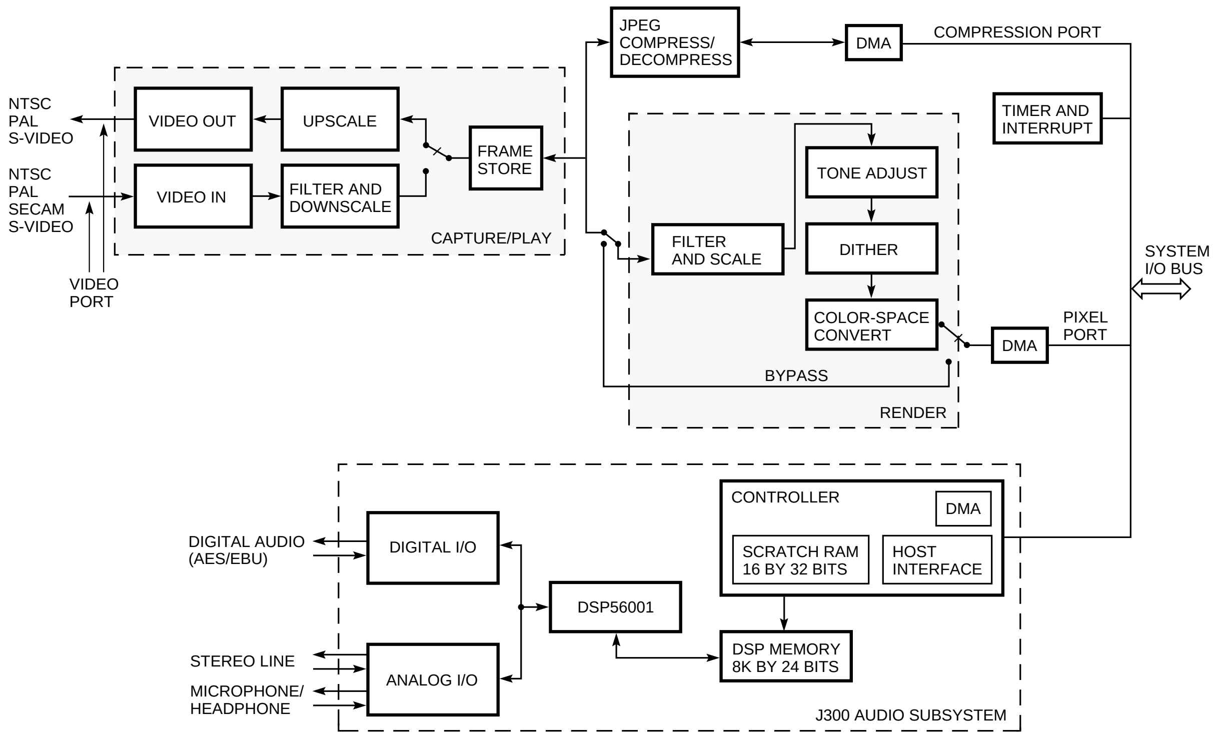


Figure 1
The Sound & Motion J300 Hardware Functional Diagram

Table 1
J300 Hardware Features

Video Subsystem	**Audio Subsystem**
Video in (NTSC, PAL, or SECAM formats)*	Compact disc (CD)-quality analog I/O
Video out (NTSC or PAL formats)	Digital I/O (AES/EBU format support)**
Composite or S-Video I/O	Headphone and microphone I/O
Still-image capture and display	Multiple sampling rates (5 to 48 kilohertz [kHz])
JPEG compression and decompression	Motorola's DSP56001 for audio processing
Image dithering	Programmable gain and attenuation
Scaling and filtering before compression	DMA into and out of system memory
Scaling and filtering before dithering	Sample counter
24-bit red, green, and blue (RGB) video out	
Two DMA channels simultaneously operable	
Video genlocking	
Graphics overlay	
150-kHz, 18-bit counter (time-stamping)	

* National (U.S.) Television System Committee, Phase Alternate Line, and Séquentiel Couleur avec Mémoire

** Audio Engineering Society/European Broadcasting Union

kernel-mode device driver, and on providing an architecture that would require the least number of system calls and minimal data copying.

The kernel-mode device driver is the lowest-level software module in the J300 software hierarchy. The driver views the J300 hardware as two distinct devices: the J300 audio and the J300 video. Depending on the requested service, the J300 kernel driver filters commands to the appropriate subsystem driver. This key decision to separate the J300 hardware by functionality influenced the design of the upper layers of the software. It allowed designers to divide the task into manageable components, both in terms of engineering effort and for project management. Separate teams worked on the two subsystems for extended periods, and the overall development time was reduced. Each subsystem had its own kernel driver, user driver, software library, test applications, and diagnostics software. The decision to separate the audio and the video software proved to be a good one. Digital's latest multimedia offering includes PCI-based FullVideo Supreme adapters that build on the video subsystem software of the J300. Unlike the J300, the newer adapters do not include an audio subsystem and thus do not use the audio library and driver.

Following the philosophy behind the actual design, the ensuing discussion of the J300 software is organized into two major sections. The first describes the software for the video subsystem, including the design and implementation of the video software library and the kernel-mode video subsystem driver. Performance data is presented at the end of this section. The second major section describes the software written for the audio subsystem. The paper then presents the methodology behind the development and testing procedures for the various software components and some improvements that are currently being investigated. A section on related published work concludes the paper.

Video Subsystem

The top of the software hierarchy for the video subsystem is the application layer, and the bottom is the kernel-mode device driver. The following simplified example illustrates the functions of the various modules that compose this hierarchy.

Consider a video application that is linked to a multimedia client library. During the course of execution, the application asks for a video operation through a call to a client library function. The client library packages the request and passes it though a socket to a multimedia server. The server, which is running in the background, picks up the request, determines the subsystem for which it is intended, and invokes the user-mode driver for that subsystem. The user-mode driver translates the server's request to an appropriate (nonblocking) video library call. Based on the operation requested, the video library builds scripts of hardware-specific commands and informs the kernel-mode device driver that new commands are available for execution on the hardware. At the next possible opportunity, the kernel driver responds by downloading these commands to the underlying hardware, which then performs the desired operation. Once the operation is complete, results are returned to the application.

Figure 2 shows a graphical representation of the software hierarchy. The modules above the kernel-mode device driver, excluding the operating system, are in user space. The remaining modules are in kernel space. The video library is modularized into device-independent and device-dependent parts. Most of the J300-specific code resides in the device-dependent portion of the library, and very little is in the kernel-mode driver. The following sections describe the various components of the video software hierarchy, beginning with the device-independent part of the video library. The description of the multimedia client library and the multimedia server is beyond the scope of this paper.

Video Library Overview

The conceptual model adopted for the software consists of three dedicated functional units: (1) capture or play, (2) compress or decompress, and (3) render or bypass. Figure 3 illustrates this model; Figure 1 shows the hardware components within each of the three

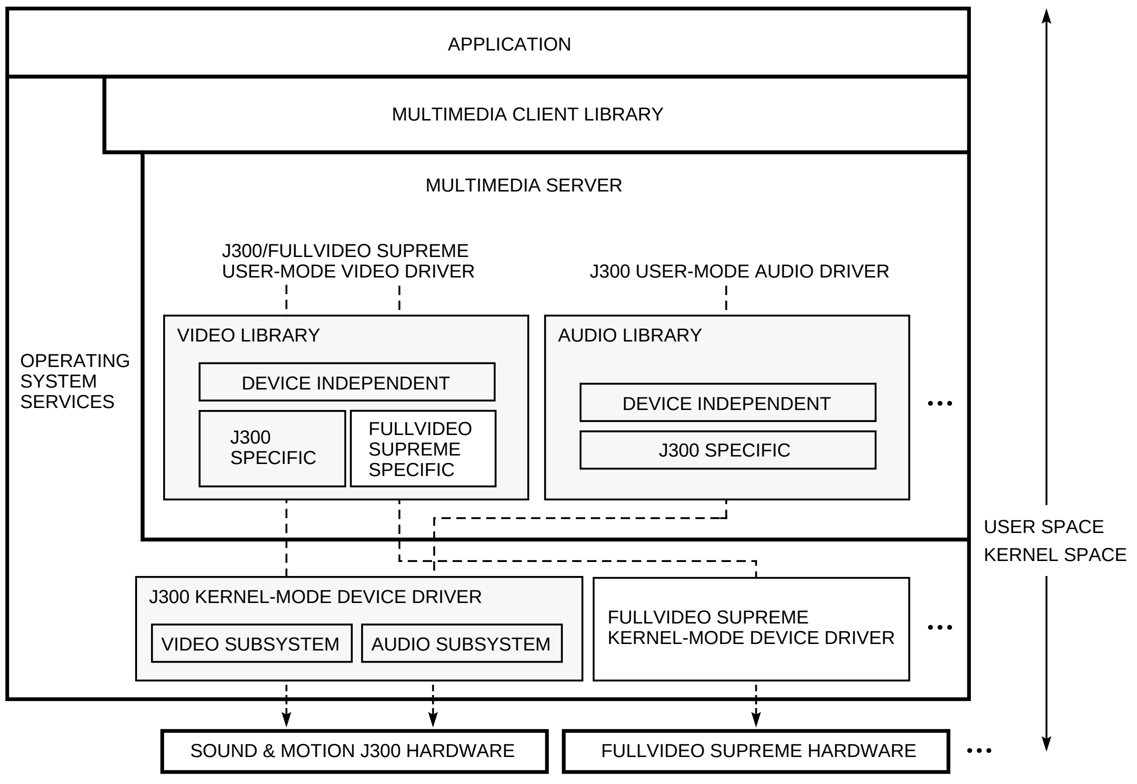

APPLICATION
MULTIMEDIA CLIENT LIBRARY
MULTIMEDIA SERVER
OPERATING SYSTEM SERVICES
J300/FULLVIDEO SUPREME USER-MODE VIDEO DRIVER
J300 USER-MODE AUDIO DRIVER
VIDEO LIBRARY
DEVICE INDEPENDENT
J300 SPECIFIC
FULLVIDEO SUPREME SPECIFIC
AUDIO LIBRARY
DEVICE INDEPENDENT
J300 SPECIFIC
USER SPACE
KERNEL SPACE
J300 KERNEL-MODE DEVICE DRIVER
VIDEO SUBSYSTEM
AUDIO SUBSYSTEM
FULLVIDEO SUPREME KERNEL-MODE DEVICE DRIVER
SOUND & MOTION J300 HARDWARE
FULLVIDEO SUPREME HARDWARE

Figure 2
The J300 Video and Audio Library as Components of Digital's Multimedia Server

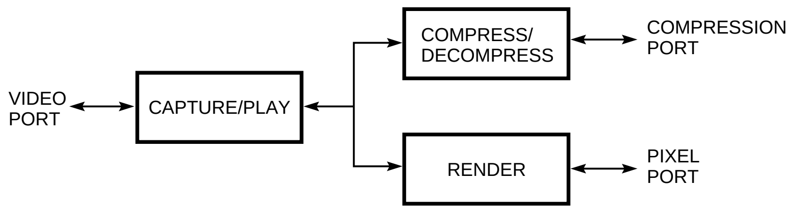


Figure 3
Conceptual Model for the J300 Video Subsystem Software

units. The units may be combined in various configurations to perform different logical operations. For example, capture may be combined with compression, or decompression may be combined with render. Figure 4 shows how these functional units can be combined to form nine different video flow paths supported by the software. Access to the units is through dedicated digital and analog ports.

All functional units and ports can be configured by the video library through tunable parameters. Algorithmic tuning is possible by configuring the three units, and I/O tuning is possible by configuring the three ports. Examples of algorithmic tuning include setting the Huffman tables or the quantization tables for the compress unit and setting the number of output colors and the sharpness for the render unit.[1,9] Examples of I/O tuning include setting the region of interest for the compression port and setting the input video format for the analog port. Thus, ports are configured to indicate the encoding of the data, whereas units are configured to indicate parameters for the video processing algorithms. Figure 5 shows the various tunable parameters for the ports and units. Figure 6 shows valid picture encoding for the two Digital I/O ports. Each functional unit operates independently on a picture. A picture is defined as a video frame, a video field, or a still image. Figure 7 illustrates the difference between a video frame and a video field. The parity setting indicates whether the picture is an even field, an odd field, or an interlaced frame.

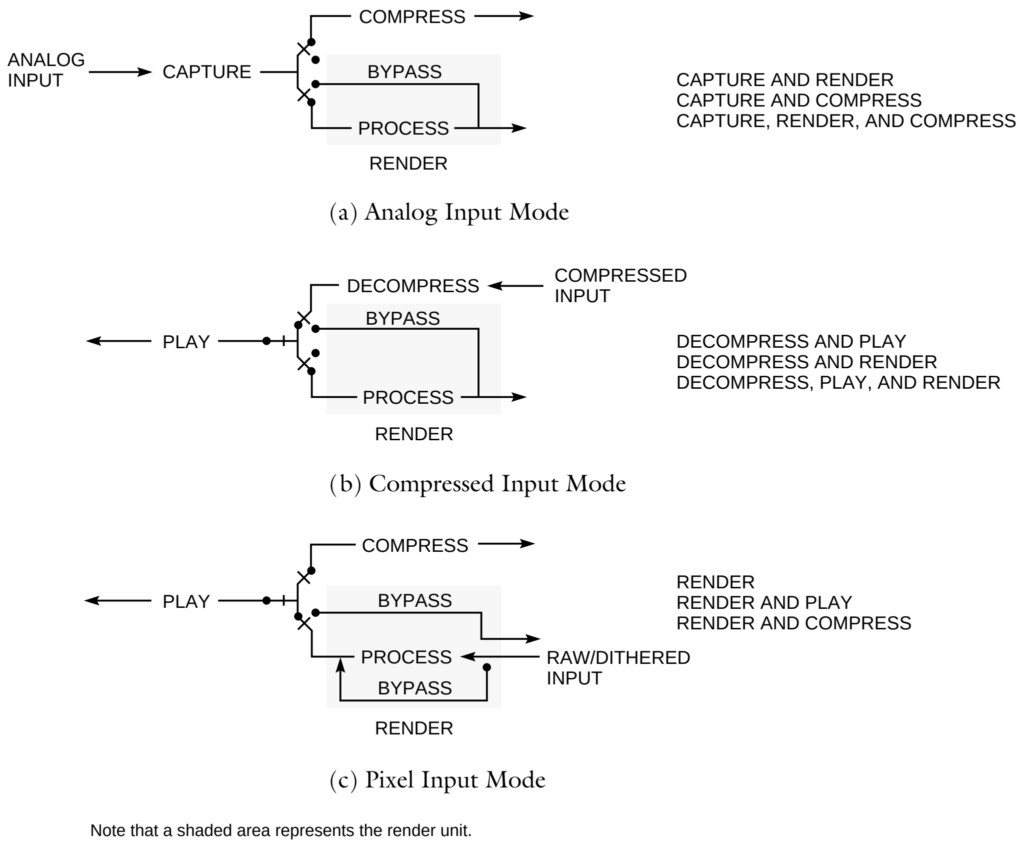


Figure 4
The Nine Different J300 Video Flow Paths

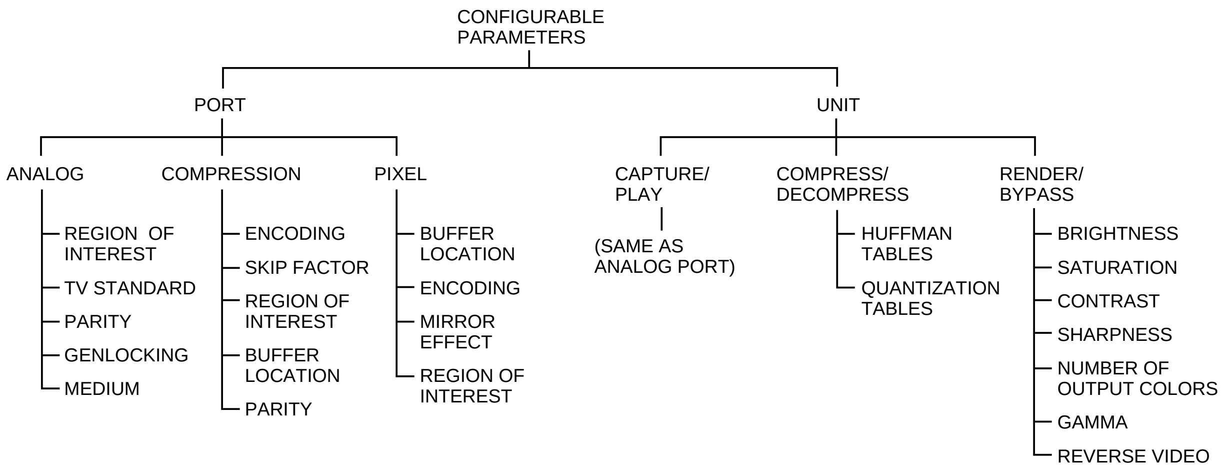


Figure 5
Tunable Parameters Provided by the J300 Video Library

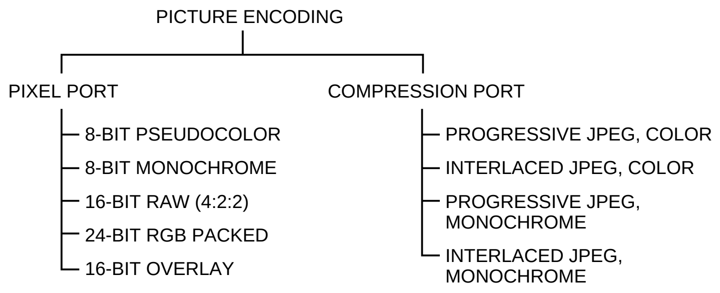


Figure 6
Valid Picture Encoding for the Two Digital I/O Ports

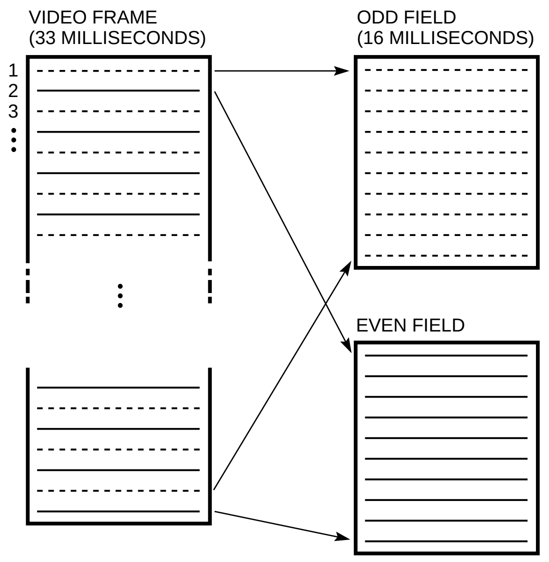


Figure 7
A Picture, Which May Be a Frame Or a Field

The software broadly classifies operations as either nonrecurring or recurring. Nonrecurring operations involve setting up the software for subsequent picture operations. An example of a nonrecurring operation is the configuration of the capture unit. Recurring operations are picture operations that applications invoke either periodically or aperiodically. Examples of recurring operations are CaptureAndCompress, RenderAndPlay, and DecompressAndRender.

All picture operations are provided in two versions: blocking and nonblocking. Blocking operations force the library to behave synchronously with the hardware, whereas nonblocking operations can be used for asynchronous program flow. Programming is simpler with blocking operations but less efficient, in terms of overall performance, as compared to nonblocking operations. All picture operations rely on combinations of input and output buffers for picture data. To avoid extra data copies, applications are required to register these I/O buffers with the library. The buffers are locked down by the library and are used for subsequent DMA transfers. Results from every picture operation come with a 90-kHz time stamp, which can be used by applications for synchronization. (The J300's 150-kHz timer is subsampled to match the timer frequency specified in the ISO MPEG-1 System Specification.)

The video library supports a client-server model of computing through the registration of parameters. In this model, the video library is part of the server process that controls the hardware. Depending on its needs, each client application may configure the hardware device differently. To support multiple clients simultaneously, the server may have to efficiently switch between the various hardware configurations. The server registers with the video library the relevant set-up parameters of the various functional units and I/O ports for each requested hardware configuration. A token returned by the library serves to identify the registered parameter sets for all subsequent operations associated with the particular configuration. Multiple clients requesting the same hardware configuration get the same token. Wherever appropriate, default values for parameters not specified during registration are used. Registrations are classified as either heavyweight, e.g., setting the number of output colors for the render unit, or lightweight, e.g., setting the quantization tables for the compress unit. A heavyweight registration often requires the library to carry out complex calculations to determine the appropriate values for the hardware and consumes more time than a lightweight registration, which may be as simple as changing a value in a register. Once set, individual parameters can be changed at a later time with edit routines provided by the library. After the client has finished using the hardware, the server unregisters the hardware configuration. The video library deletes all related internal state information associated with that configuration only if no other client is using the same configuration.

The library provides routines for querying the configurations of the ports and units at any given time. Extensive error checking and reporting are built into the software.

Video Library Operation

Internally, the video library relies on queues for supporting asynchronous (nonblocking) flow control and for obtaining good performance. Three types of queues are defined within the library: (1) command queue, (2) event (or status) queue, and (3) request queue. The command and event queues are allocated by the kernel-mode driver from the nonpaged system memory pool at kernel-driver load time. At device open time, the two queues are mapped to the user virtual memory address space and subsequently shared by the video library and the kernel-mode driver. The request queue, on the other hand, is allocated by the library at device open time and is part of the user virtual memory space. Detailed descriptions of the three types of queues follow. An example shows how the queues are used.

Command Queue The command queue, the heart of the library, is employed for one-way communication from the library to the kernel driver. Figure 8 shows the composition of the command queue. Essentially, the command queue contains commands that set up, start, and stop the hardware for picture operations. Picture operations correspond to video library calls invoked by the user-mode driver. Even though the architecture does not impose any restrictions, a picture operation usually consists of two scripts: the first script sets up the operation, and the second script cleans up after the hardware completes the operation. Scripts are made up of packets. The header packet is called a script packet, and the remaining packets are called command packets. The library builds packets and puts them into the command queue. The kernel driver retrieves and interprets script packets and downloads the command packets to the hardware. Script packets provide the kernel driver with information about the type of script, the number of command packets that constitute the script, and the hardware interrupt to expect once all command packets have been downloaded. Command packets are register I/O operations. A command packet can contain the type of register access desired, the kernel virtual address of the register, and the value to use if it is a write operation. The library uses identifiers associated with the command packets and the script packets to identify the associated operation. The command queue is managed as a ring buffer. Two indexes called PUT and GET dictate where new packets get added and from where old packets are to be extracted. A first-in, first-out (FIFO) service policy is adhered to. The library manages the PUT index, and the kernel driver manages the GET index.

Event Queue The event queue, a companion to the command queue, is also used for one-way communication but in the reverse direction, i.e., from the kernel driver to the library. Figure 9 shows the composition of the event queue. The kernel driver puts information into the queue in the form of event packets whenever a hardware interrupt (event) occurs. Event packets contain the type of hardware interrupt, the time at which the interrupt occurred, an integer to identify the completed request, and, when appropriate, a value from a relevant hardware register. The library monitors the queue and examines the event packets to determine which requested picture operation completed. As is the case with the command queue, the event queue is managed as a ring buffer with a FIFO service policy. The library manipulates the GET index, and the kernel driver manipulates the PUT index.

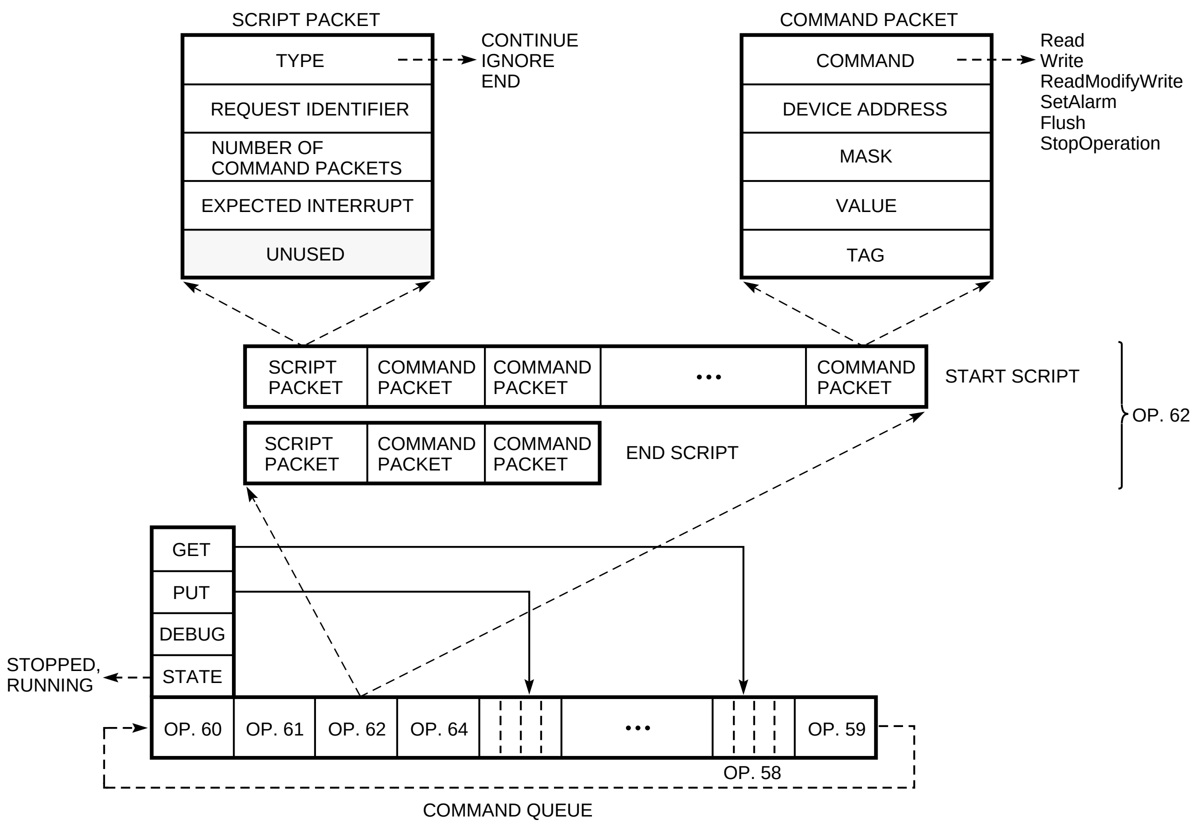


Figure 8
The Command Queue

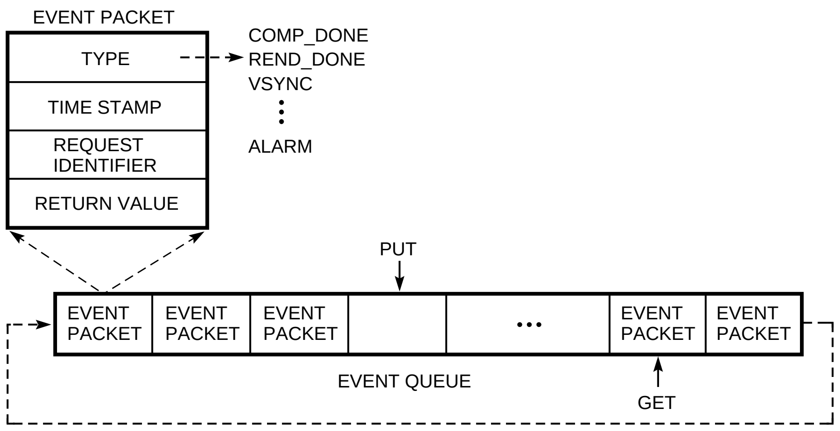


Figure 9
The Event Queue

Request Queue The library uses the request queue to coordinate user-mode driver requests with operations in the command queue and with completed events in the event queue. When a picture operation is requested, the library builds a request packet and places it in the request queue. The packet contains all information relevant to the operation, such as the location of the source or destination buffer, its size, and scatter/gather maps for DMA. Subsequently, the library uses the request packet to program the command queue. Once the operation has completed, the associated request packet provides the information that the library needs for returning the results to the user-mode driver. As with the other queues, the service policy is FIFO, and the queue is managed as a ring buffer.

Capture and Render Example Figure 10 shows an application displaying live video on a UNIX workstation that contains a J300 adapter. The picture operation that makes this possible is the video library's CaptureAndRender operation. A description of the asynchronous flow of control when the user-mode driver invokes a CaptureAndRender picture operation follows. This example illustrates the typical interaction between the various software and hardware components. The discussion places special emphasis on the use of the queues previously described.

1. The user-mode video driver invokes a nonblocking CaptureAndRender picture operation with appropriate arguments.
2. The library builds a request packet, assigns an identifier to it, and adds the packet to the request queue. Subsequently, it builds the script and command packets needed for setting up and terminating the operation and adds them to the command queue. It then invokes the kernel driver's start I/O routine, to indicate that new hardware scripts have been added to the command queue.
3. Start I/O queues up the kernel routine (which downloads the command scripts to the hardware) in the operating system's internal call-out queue as a deferred procedure call (DPC) and returns control to the video library.[10]

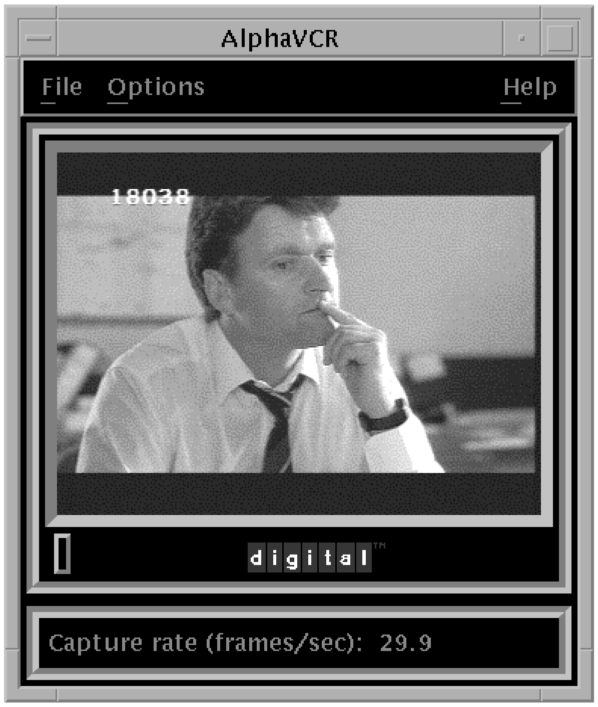


Figure 10
Live Video on a UNIX Workstation Using the Capture and Render Path

4. The video library returns control to the user-mode driver, which continues from where it had left off, performing other tasks until it invokes a blocking (i.e., wait) routine. This gives the library an opportunity to check the event queue for new events. If there are no new events to service, the library asks the kernel driver to "put it to sleep" until a new event arrives.
5. In the meantime, the DPC that had previously been queued up starts to execute after being invoked by the operating system's scheduler. The job of the DPC is to read and interpret script packets and, based on the interpretation, to download the command packets that constitute the script. Only the first script that sets up and starts the operation is downloaded to the hardware.
6. A hardware interrupt signaling the completion of the operation occurs, and control is passed to the kernel driver's hardware interrupt service routine (ISR). The hardware ISR clears the interrupt line, logs the time, and queues up a software ISR in the system's call-out queue, passing it relevant information such as the interrupt type and an associated time stamp.
7. The operating system's scheduler invokes the queued software ISR. The ISR then reads and interprets the current (end) script packet in the command queue, which provides the type of interrupt to expect as a result of downloading the previous (start) script. The software ISR checks to see if the interrupt that was passed to it is the same as one that was predicted by the (end) script. For example, a script that starts a render operation may expect to see a REND_DONE event. When the actual event matches the predicted event, the command packets associated with the current (end) script are downloaded to the hardware.
8. After all command packets from the (end) script have been downloaded, the software ISR logs the type of event, the associated time stamp, and an identifier for the completed operation into the event queue. It then issues a wake-up call to any "sleeping" or blocked operations that might have been waiting for hardware events.
9. The system wakes the sleeping library routine, which checks the event queue for new events. If a REND_DONE event is present, the library uses the request identifier from the event packet to get the associated request packet from the request queue. It then places the results of the operation in the memory locations that are pointed to by addresses in the request packet and that belong to the user-mode driver. (The buffer containing the rendered data is not copied because it already belongs to the user-mode driver.) The library updates the GET

indexes of the event and request queues and returns control to the user-mode driver.

10. The user-mode driver may then continue to queue up more operations.

Figure 11 shows a graphical representation of the capture and render example. If desired, multiple picture operations can be programmed through the library before a single one is downloaded by the driver and executed by the hardware. Additionally, performance is enhanced by improving the asynchronous flow through the use of multiple buffers for the different functional units shown in Figure 3.

Sometimes it is necessary to bypass the queuing mechanism and program the hardware directly. This is especially true for hardware diagnostics and operations such as hardware resetting, which require immediate action. In addition, for slow operations, such as setting the analog port (video-in circuitry), programming the hardware in the kernel using queues is undesirable. The kernel driver supports an immediate mode of operation that is accomplished by mapping the hardware to the library's memory space, disabling the command queue, and allowing the library to program the hardware directly.

The Kernel-mode Video Driver

To keep the complexity of the kernel-mode video driver manageable, we made a clear distinction between device programming and device register loading. Device-specific programming is done in user space by the video library; device register I/O (without contextual understanding) is performed by the kernel driver. Separating the tasks in this manner resulted in a kernel driver that incorporates little device-specific knowledge and thus is easily portable across multiple devices.

The kernel driver allows only one process to access the device at any particular time. (Support for multiple-process access is provided by the multimedia server.) Components of the video kernel-mode driver include

- An Initialization Routine—The driver's initialization routine is executed by the operating system at driver load time. The primary function of this routine is to reserve system resources such as nonpaged kernel memory for the command queue, the event queue, and the other internal data structures needed by the driver.
- A Set of Dispatch Routines—Dispatch routines constitute the main set of static functionality provided by the driver. The driver provides dispatch routines for opening and closing the video subsystem, for mapping and unmapping hardware registers to the kernel and to user virtual memory address spaces, for locking and unlocking noncontiguous memory for scatter/gather DMA, and for mapping and unmapping the various queues to the library.
- An Asynchronous I/O Routine—The video library invokes this routine to check for pending events that have to be processed. If an unserviced event exists, the kernel driver immediately returns control to the library; if no event exists, the system puts the library process to sleep.
- A Start I/O Routine and a Stop I/O Routine—The driver uses the start I/O routine to initiate data

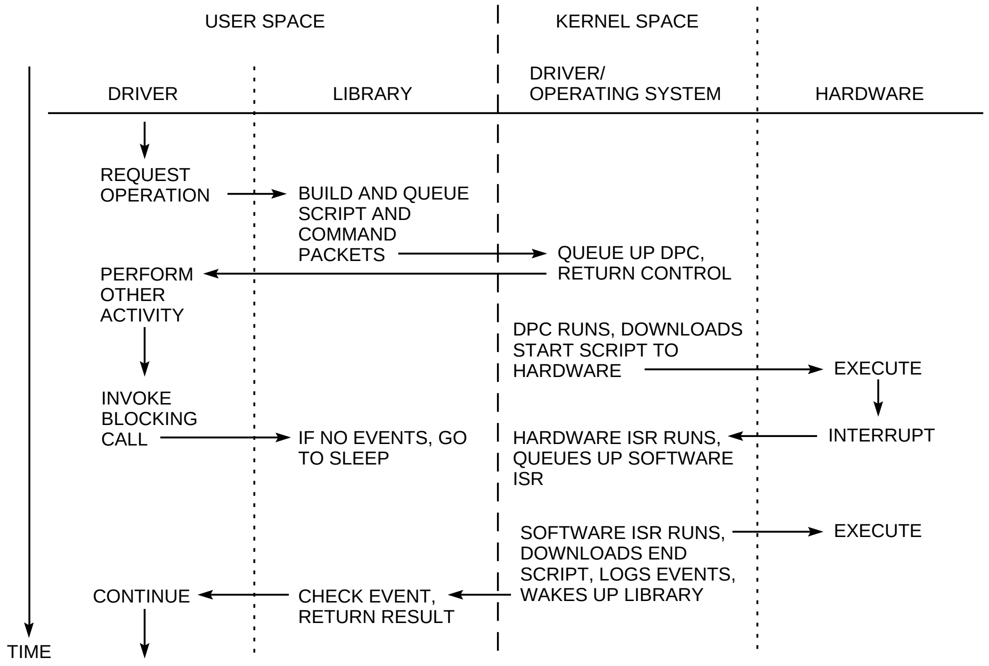

USER SPACE
KERNEL SPACE
DRIVER
LIBRARY
DRIVER/ OPERATING SYSTEM
HARDWARE
REQUEST OPERATION
BUILD AND QUEUE SCRIPT AND COMMAND PACKETS
QUEUE UP DPC, RETURN CONTROL
PERFORM OTHER ACTIVITY
DPC RUNS, DOWNLOADS START SCRIPT TO HARDWARE
EXECUTE
INVOKE BLOCKING CALL
IF NO EVENTS, GO TO SLEEP
HARDWARE ISR RUNS, QUEUES UP SOFTWARE ISR
INTERRUPT
SOFTWARE ISR RUNS, DOWNLOADS END SCRIPT, LOGS EVENTS, WAKES UP LIBRARY
EXECUTE
CONTINUE
CHECK EVENT, RETURN RESULT
TIME

Figure 11
One Case of Simplified Flow Control When Using the Video Subsystem

transfers to and from the J300 by downloading register I/O commands from the command queue to the J300. The stop I/O routine is used to terminate the downloading of future scripts. For performance reasons, scripts in the process of being downloaded cannot be stopped.

- A Hardware Interrupt Service Routine—Since the hardware ISR runs at a higher priority than both system and user space routines, it has purposely been kept small, performing only simple tasks that are absolutely necessary and time critical. Specifically, the hardware ISR records the interrupt and the time at which it occurred. It then clears the interrupt and queues up a software ISR.
- A Software Interrupt Service Routine—The software ISR is the heart of the kernel driver. It runs at a lower interrupt request level (IRQL) than the hardware ISR but has a higher priority than user-space routines. The software ISR is invoked as a DPC either by the hardware ISR or by the library through a start I/O request. Its main function is to process script packets and download command packets programmed by the video library.

Debugging the Video Subsystem

Because of the real-time nature of operations, debugging the software was a challenge. The size of the code, the complex interaction between the various functional pieces, and the asynchronous nature of operations suggested that, for debugging purposes, it would be helpful if hardware commands could be scrutinized just before the final downloading took place. Fortunately, the video library's extensive use of queues made it possible for us to design a custom tool with knowledge of the hardware and software architectures that would allow us to examine the command scripts.

In addition to presenting a debugging challenge, the real-time nature of operations limited the scope of UNIX tools like dbx, kdbx, and ctrace. Timing was important, and the debugger had the tendency to slow down the overall program to the point where a previous failure on a free system would not occur with the debugger enabled. To catch some of these elusive bugs while preserving the timing integrity of the operations, the scratch random-access memory (RAM) on the J300 audio subsystem (see Figure 1) was used to store traces. A brief description of the two approaches follows.

Queue Interpreter The queue interpreter was specifically developed as an aid for debugging the video library. As the name suggests, its primary function was to interpret the commands in the command queue and the events in the event queue. At random locations in the library, a list of hardware commands currently in the command queue could be viewed before the kernel driver downloaded them for execution. For each command, the information displayed included a sequence number, the type of operation, the ASCII name of the register to be accessed, the register's physical address, the value to be written, and, when possible, a bit-wise interpretation of the value. This information was used to check if the upper layer software had programmed the device registers in the correct sequence and with the proper values.

Another important capability of the queue interpreter was that it could step through the command packets and download each command separately. On many occasions, this function helped locate and isolate the specific register access that was causing the hardware to stall or to crash the system. By using the sequence number, the offending hardware command could be traced to the precise location in the library where it had been programmed.

In addition, the queue interpreter was able to search the command queue for any access to a specific hardware register, could display the contents of the event queue, and had a "quiet mode," in which the interpreter would log the commands on a disk for later analysis.

Audio RAM Printer Although it was a useful tool for debugging, the queue interpreter was not a good real-time tool because it slowed down the overall program execution and thus affected the actual timing. Similarly, kernel driver operations could not be traced using the system's printf( ) command because it too affected the timing. Furthermore, because of the asynchronous nature of printf( ) and the possibility of losing it, printf( ) was ineffective in pinpointing the precise command that had caused the system to fail. Thus, we had to find an alternate mechanism for debugging failures related to timing.

The J300 audio subsystem has an 8K-by-24-bit RAM that is never used for any video operations. This observation led to the implementation of a print function that wrote directly to the J300's audio RAM. This modified print function was intermixed in the suspect code fragment in the kernel driver to facilitate trace analysis. When a system failure occurred or after the application had stopped, a companion "sniffer" routine would read and dump the contents of the RAM to the screen or to a file for analysis. The modified print function was used primarily for debugging dynamic operations such as the ones in the hardware and software interrupt handlers. Many bugs were found and fixed using this technique. The one caveat was that this technique was useful only in cases where the video subsystem was causing a system failure independent of the operation of the audio subsystem.

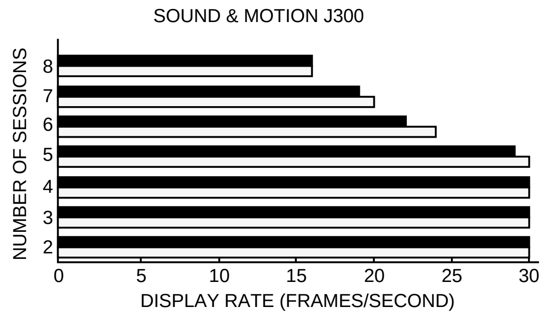
Video Subsystem Performance

Measuring the true performance of any software is generally a difficult task. The complex interaction between different modules and the number of variables that must be fixed makes the task arduous. For video, the problem is aggravated by the fact that the speed with which the underlying video compression algorithm works is nonlinearly dependent on the content of the video frames and the desired compression ratio. A user working with a compressed sequence that contains images that are smooth (i.e., have high spatial redundancy) will get a faster decompression rate than a user who has a sequence that contains images that have regions of high frequencies (i.e., have low spatial redundancy). A similar discrepancy will exist when sequences with different compression ratios are used. Since there are no standard video sequences available, the analyst has to make a best guess at choosing a set of representative sequences for experiments. Because the final results are dependent on the input data, they are influenced by this decision. Other possible reasons for the variability of results are the differing loads on the operating systems, the different configurations of the underlying software, and the overhead imposed by the different test applications.

Our motivation for checking the performance of the J300 and FullVideo Supreme JPEG adapters was to determine whether we had succeeded in our goal of developing software that would extract real-time performance while adding minimal overhead. The platforms we used in our experiments were the AlphaStation 600 5/266 and the DEC 3000 Model 900. The AlphaStation 600 5/266 was chosen because it is a PCI-based system and could be used to test the FullVideo Supreme JPEG adapter. The DEC 3000 Model 900 is a TURBOchannel system and could be used to test the J300 adapter. Both systems are built around the 64-bit Alpha 21064A processor running at clock rates of 266 megahertz (MHz) and 275 MHz, respectively. Each system was configured with 256 megabytes of physical memory, and each was running the Digital UNIX Version 3.2 operating system and Digital's Multimedia Services Version 2.0 for Digital UNIX software. No compute-intensive or I/O processes were running in the background, and, hence, the systems were lightly loaded.

Our experiments were designed to reflect real applications, and special emphasis was placed on obtaining reproducible performance data. The aim was to understand how the performance of individual sessions was affected as the number of video sessions was increased. We wrote an application that captured, dithered, and displayed a live video stream obtained from a camera while simultaneously decompressing, dithering, and displaying multiple video streams read from a local disk. This is a common function in teleconferencing applications where the multiple compressed video streams come over the network. We measured the display rate for the video sequence that was being captured and dithered and the average display rate for sequences that were being decompressed and dithered. The compressed sequences had an average peak signal-to-noise ratio (PSNR) of 27.8 decibels (dB) and an average compression ratio of approximately 0.6 bits per pixel. The sequences had been compressed and stored on the local disk prior to the experiment. Image frame size was source input format (SIF) 352 pixels by 240 lines. Figure 12 and Figure 13 illustrate the performance data obtained as a result of the experiments.

In general, we were satisfied with the performance results. As seen in Figures 12 and 13, a total of five sessions can be accommodated at 30 frames per second with the J300 on a DEC 3000 Model 900 system and three sessions at 30 frames per second with the



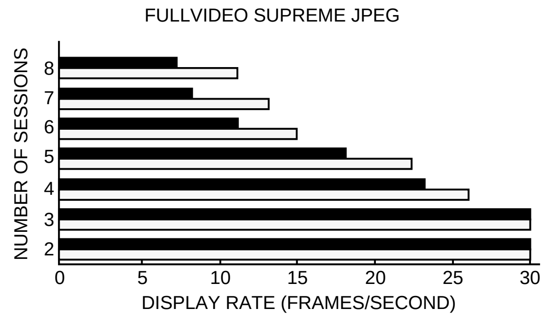
KEY:

□ DECOMPRESS, DITHER, AND DISPLAY

■ CAPTURE, DITHER, AND DISPLAY

Note: The number of sessions *n* is equal to one capture plus (*n* – 1) decompressions.

Figure 12
Performance Data Generated by a DEC 3000 Model 900 System with a Sound & Motion J300 Adapter



KEY:

□ DECOMPRESS, DITHER, AND DISPLAY

■ CAPTURE, DITHER, AND DISPLAY

Note: The number of sessions *n* is equal to one capture plus (*n* – 1) decompressions.

Figure 13
Performance Data Generated by an AlphaStation 600 5/266 with a FullVideo Supreme JPEG Adapter

FullVideo Supreme JPEG on an AlphaStation 600 5/266 system. The discrepancy in performance of the two systems may be attributed to the differences in CPU, system bus, and maximum burst length. The DEC 3000 Model 900 has a 32-bit TURBOchannel bus whose speed is 40 nanoseconds with a peak transfer rate of 100 megabytes per second, whereas the AlphaStation 600 5/266 has a PCI bus whose speed is 30 nanoseconds. The DMA controller on the J300 adapter has a maximum burst length of 2K pages, whereas the FullVideo Supreme JPEG adapter has a maximum burst length of 96 bytes. Since in our experiments data was dithered and sent over the bus (at 83 Kbytes per frame) to the frame buffer, burst length becomes the dominant factor, and it is not unreasonable to expect the J300 to perform better than the FullVideo Supreme JPEG.

The difference between capture and decompression rate (as shown in Figures 12 and 13) may be explained as follows: Decompression operations are intermixed between capture operations, which occur at a frequency of one every 33 milliseconds. Overall performance improves when a larger number of decompression operations are accommodated between successive capture operations. Since the amount of time the hardware takes to decompress a single frame is unknown (the time depends on the picture content), the software is unable to determine the precise number of decompression operations that can be programmed. Also, in the present architecture, since all operations have equal priority, if a scheduled decompression operation takes longer than expected, it is liable to not relinquish the hardware when a new frame arrives, thus reducing the capture rate. When we ran the decompression, dither, and display operation only (with the capture operation turned off), the peak rate achieved by the FullVideo Supreme JPEG adapter was approximately 165 frames per second, and the rate for the Sound & Motion J300 was about 118 frames per second. Bus speed and hardware enhancements in the FullVideo Supreme JPEG can be attributed to the difference in the two rates.

The next section describes the architecture for the J300 audio subsystem. Relative to the video subsystem, the audio software architecture is simpler and took less time to develop.

Audio Subsystem

The J300 audio subsystem complements the J300 video subsystem by providing a rich set of functional routines by way of an audio library. The software hierarchy for the audio subsystem is similar to the one for the video subsystem. Figure 2 shows the various components of this hierarchy as implemented under the Digital UNIX operating system. Briefly, an application makes a request to a multimedia server for processing audio. The request is made through invocation of routines provided by a multimedia client library. The multimedia server parses the request and dispatches the appropriate user-mode driver, which is built on top of the audio library. Depending on the request, the audio library may perform the operation either on the native CPU or alternatively on the J300 digital signal processor (DSP). Completed results are returned to the application using the described path in the reverse direction.

To provide a comprehensive list of audio processing routines, the software relies on both host-based and J300-based processing. The workhorse of the J300 audio subsystem is the general-purpose Motorola Semiconductor DSP56001 (see Figure 14), which provides hardware control for the various audio components while performing complex signal processing tasks at real-time rates. Most notable, software running on the DSP initiates DMA to and from system memory, controls digital (AES/EBU) audio I/O, manages analog stereo and mono I/O, and supports multiple sampling rates, including Telephony (8 kHz) and fractions of digital audio tape (DAT) (48 kHz) and compact disc (CD) (44.1 kHz) rates. The single-instruction multiply, add, and multiply-accumulate operations, the two data moves per instruction operations, and the low overhead for specialized data addressing make the DSP

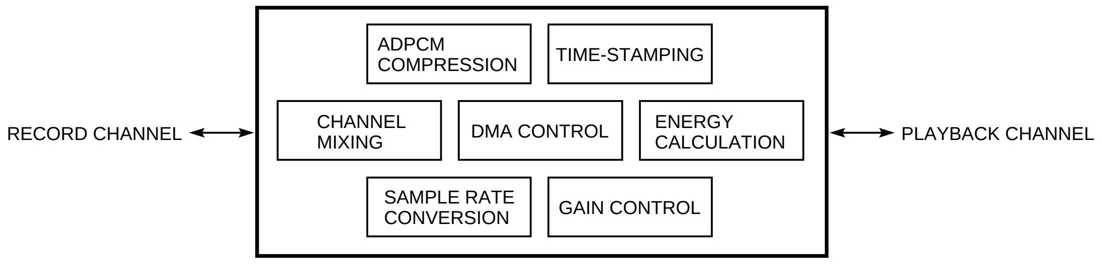


Figure 14
Some Audio Functions Supported by Motorola's DSP56001 Processor

especially suitable for compute-intensive audio processing tasks. Real-time functions such as adaptive differential pulse code modulation (ADPCM) encoding and decoding, energy calculation, gain control for analog-to-digital (A/D) and digital-to-analog (D/A) converters, and time-stamping are performed by software running on the DSP.[11] Other tasks such as converting between different audio formats (μ-law, A-law, and linear), mixing and unmixing of multiple audio streams, and correlating the system time with the J300 90-kHz timer and with the sample counter are done on the native CPU by the library software.[12]

Early in the project, we had to decide whether or not to expose the DSP to the client applications. Exposing the DSP would have provided additional flexibility for application writers. Although this was an important reason, the opposing arguments, which were based on the negative consequences of exposing the raw hardware, were more compelling. System security and reliability would have been compromised; an incorrectly programmed DSP could cause the system to fail and could corrupt the kernel data structures. Additionally, maintaining, debugging, and supporting the software would be difficult. To succeed, the product had to be reliable. Therefore, we decided to retain control of the software but to provide enough flexibility to satisfy as many application writers as possible. As customer demand and feedback grew, more DSP programs would be added to the list of existing programs in a controlled manner to ensure the integrity and robustness of the system.

The following subsections describe the basic concepts behind the device-independent portion of the audio library and provide an operational overview of the library internals.

Audio Library Overview

The audio library defines a single audio sample as the fundamental unit for audio processing. Depending on the type of encoding and whether it is mono or stereo, an audio sample may be any of the following: a 4-bit ADPCM code word, a pair of left/right 4-bit ADPCM code words, a 16-bit linear pulse code modulation (PCM) audio level, a pair of left/right 16-bit linear PCM audio levels, an 8-bit μ-law level, or an 8-bit A-law level. The library defines continually flowing audio samples as an audio stream whose attributes can be set by applications. Attributes provide information on the sampling rate, the type of encoding, and how to interpret each sample.

Audio streams flow through distinct directional virtual channels. Specifically, an audio stream flows into the subsystem for processing through a record (input) channel, and a processed stream flows out of the subsystem through a playback (output) channel. A configurable bypass mode in which the channels are used for a direct path to the hardware I/O ports is also provided. As is the case for audio streams, each channel has attributes such as a buffer for storing captured data, a buffer for storing data to be played out, permissions for channel access, and a sample counter. Sample counters are used by the library to determine the last audio sample processed by the hardware. Channel permissions determine the actions allowed on the channel. Possible actions include read, write, mix, unmix, and gain control or combinations of these actions.

The buffers associated with the I/O channels are for queuing unserviced audio data and are called smoothing buffers. A smoothing buffer ensures a continuous flow of data by preventing samples from being lost due to the non-real-time scheduling by the underlying operating system. The library provides nonblocking routines that can read, write, mix, and unmix audio samples contained in the channel buffers. A sliding access window determines which samples can be accessed within the buffer. The access window is characterized in sample-time units, and its size is proportional to that of the channel buffer that holds the audio data.

Like the video library, the audio library supports multiple device configurations through a set of registration routines. Clients may register channel and audio stream parameters with the library (through the server) at set-up time. Once registered, the parameters can be changed only by unregistering and then reregistering. The library provides query routines that return status/progress information, including the samples processed, the times (both system and J300 specific) at which they were processed, and the channel and stream configurations. Overall, the library supports four operational (execution) modes: teleconferencing, compression, decompression, and rate conversion. Extensive error checking and reporting are incorporated into the software.

Audio Library Operation

The execution mode and the associated DSP program dictate the operation of the audio library. Execution modes are user selectable. All programs support multiple sampling rates, I/O gain control, and start and pause features, and provide location information for the sample being processed within the channel buffer. Buffers associated with the record and playback channels are treated as ring buffers with a FIFO service policy. Management of data in the buffers is through integer indexes (GET and PUT) using an approach similar to the one adopted for the management of the command and event queues in the video subsystem. Specifically, the DMA controller moves the audio data from the DSP's external memory to the area in the channel buffer (host memory) starting at the PUT index. Audio data in this same channel buffer is pulled by the host (library) from the location pointed to by the GET index. Managers of the GET and PUT indexes are reversed when DMA is being performed from a channel buffer to the DSP external memory. In all cases, the FIFO service policy ensures that the audio data is processed in the sequence in which it arrives.

The internal operation of the audio library is best explained with the help of a simple example that captures analog audio from the J300 line-in connector and plays out the data through the J300's line-out connector. This most basic I/O operation is incorporated in more elaborate audio processing programs. The example follows.

1. The server opens the audio subsystem, allocates memory for the I/O buffers, and invokes a library routine to lock down the buffers. Two buffers are associated with the record and playback channels.
2. The library sets up the DSP external memory for communications between software running on the two processors, i.e., the CPU and the DSP. The set-up procedure involves writing information at locations known and accessible to both processors. The information pertains to the physical addresses needed by the DMA scheduler portion of the DSP program and for storing progress information.
3. A kernel driver routine maps a section of system memory to user space. This shared memory is used for communication between the driver and the library. The type of information passed back and forth includes the sample number being processed, the associated time stamps, and the location of the GET and PUT indexes within the I/O buffers.
4. Other set-up tasks performed by the library include choosing the I/O connectors, setting the gain for the I/O channels, and loading the appropriate DSP program. A start routine enables the DSP.
5. Once the DSP is enabled, all components in the audio hardware are under its control. The DSP programs the DMA controller to take sampled audio data from the line-in connector and move it into the record channel buffer. It then programs the same controller to grab data from the playback channel buffer and move it to the external memory from where it is played out on the line-out connector.
6. The library monitors the indexes associated with the I/O buffers to determine the progress, and, based on the index values, the application copies data from the input channel to the output channel buffer. The access window ensures that data copying stays behind the DSP, in the case of input, and in front of the DSP, in the case of output.

Support for Multiple Adapters

The primary reason for using multiple J300 adapters is to overcome the inherent limitations of using a single J300. First, a single J300 limits the application to a single video port and a single audio input port. Some applications process multiple video input streams simultaneously. For example, a television station receiving multiple video feeds may want to compress and store these for later usage utilizing a single workstation. Another example is the monitoring of multiple video feeds from strategically placed video cameras for the purpose of security. Since AlphaStation systems have the necessary horsepower to process several streams simultaneously, supporting multiple J300s on the same system is desirable.

Second, if a single J300 is used, the video-in and video-out ports cannot be used simultaneously. This limitation exists because the two ports share a common frame store, as shown in Figure 1, and programming the video-in and video-out chip sets is a heavyweight operation. Multiple J300s can alleviate this problem. One example of an application that requires the simultaneous use of the video-in and video-out ports is a teleconferencing application in which the video-in circuitry is used for capturing the camera output, and the video-out circuitry is used for sending regular snapshots of the workstation screen to an overhead projection screen. A second example is an application that converts video streams from one format to another (e.g., PAL, SECAM, NTSC) in real time.

As a result of the limitations just cited, support for multiple J300s on the same workstation was one of the project's design goals. In terms of coding, achieving this goal required not relying on global variables and using indexed structures to maintain state information. Also, because of the multithreaded nature of the server, care had to be taken to ensure that data and operation integrity was maintained.

For most Alpha systems, the overall performance remains good even with two J300s on the same system. For high-end systems, up to three J300s may be used. The dominant limitation in the number of J300s that can be handled by a system is the bus bandwidth. As the number of J300s in the system increases, the data traffic on the system bus increases proportionally.

Having described the software architecture, we now shift our attention to the development environment, testing strategy, and diagnostics software.

Software Development Environment

During the early phases of the development process, we depended almost exclusively on Jvideo. Since the J300 is primarily a cost-reduced version of Jvideo, we were able to develop, test, and validate the design of the device-independent portion of the software and most of the kernel device driver well before the actual J300 hardware arrived. Our platform consisted of a Jvideo attached to a DECstation workstation, which was based on a MIPS R3000 processor and was running the ULTRIX operating system. When the new Alpha workstations became available, we switched our development to these newer and faster machines. We ported the 32-bit software to Alpha's 64-bit architecture. Sections of the kernel device driver were rewritten, but the basic design remained intact. The overall porting effort took a little more than a month to complete. At the end of that time, we had the software running on a Jvideo attached to an Alpha workstation, which was running the DEC OSF/1 operating system (now called the Digital UNIX operating system). We promptly corrected software timing bugs exposed as a result of using the fast Alpha-based workstations.

For the development of the device-dependent portion, we relied on hardware simulation of the J300. The different components and circuits of the J300 were modeled with Verilog behavioral constructs. Accesses to the TURBOchannel bus were simulated through interprocess communication calls (IPCs) and shared memory (see Figure 15). Because a 64-bit version of Verilog was unavailable, simulations were run on a machine based on the MIPS R3000 processor running the ULTRIX operating system. The process, though accurate, was generally slow.

Testing and Diagnostics

We wrote several applications to test the software architecture. The purpose of these applications was to test the software features in real-world situations and to demonstrate through working sample code how the libraries could be used. Applications were classified as video only, audio only, and ones that contained both video and audio.

In addition, we wrote two types of diagnostic software to test the underlying hardware components: (1) read-only memory (ROM) based and (2) operating system based. ROM-based diagnostics have the advantage that they can be executed from the console level without first booting the system. The coverage provided is limited, however, because of the complexity of the hardware and the limited size of the ROM. Operating system diagnostics rely on the kernel device driver and on some of the library software. This suite of tests provides comprehensive coverage with verifications of all the functional blocks on the J300. For the new PCI-based FullVideo Supreme video adapters, only operating-system-based diagnostics exist.

Related Work

When the Jvideo was conceived in early 1991, little had been published on hardware and software solutions for putting video on the desktop. This may have been partly due to the newness of the compression standards and to the difficulty in obtaining specialized video compression silicon. Since then, audio and video compression have become mainstream, and several computer vendors now have products that add multimedia capability to the base workstations.

Lee and Sudharsanan describe a hardware and software design for a JPEG microchannel adapter card built for platforms based on IBM's PS/2 operating system.[13] The adapter is controlled by an interrupt-driven software running under DOS. In addition, the software is also responsible for color-space conversion and algorithmic tuning of the JPEG parameters. Audio support is not included in the hardware. The paper presents details on how the software programs the various components of the board (e.g., the CL550 chip from C-Cube Microsystems and the DMA logic) to achieve compression and decompression. Portability of the software is compromised since the bulk of the

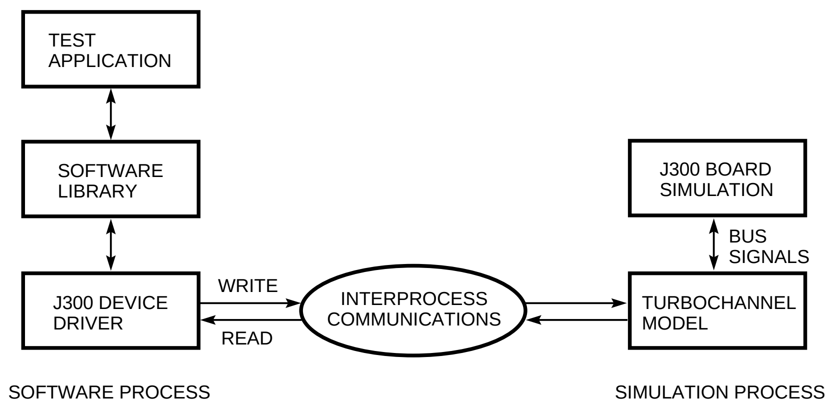


Figure 15
Hardware Simulation Environment for Software Development

code, which resides inside the interrupt service routine, is written in assembly language.

Boliek and Allen describe the implementation of hardware that, in addition to providing baseline JPEG compression, uses a dynamic quantization circuit to achieve fixed-rate compression.[14] The board is based on the RIOH JPEG chip set that includes separate chips for performing the DCT, Huffman coding, and color-space conversion. The paper's main focus is on describing the Allen Parameterized (orthogonal) Transform that approximates the DCT while reducing the cost of the hardware. The paper contains little information about software control, architecture, and control flow.

Traditionally, operating systems have relied on data copying between user space and kernel space to protect the integrity of the kernel. Although this method works for most applications, for multimedia applications, which usually involve massive amounts of data, the overhead of data copying can seriously compromise the system's real-time performance.[15] Fall and Pasquale describe a mechanism of in-kernel data paths that directly connect the source and sink devices.[16] Peer-to-peer I/O avoids unnecessary data copying and improves system and application performance. Kitamura et al. describe an operating system architecture, which they refer to as the zero-copy architecture, that is also aimed at reducing the overhead due to data copying.[17] The architecture uses memory mapping to expose the same physical addresses to both the kernel and the user-space processes and is especially suitable for multimedia operations. The J300 software is also a zero-copy architecture. No data is copied between system and user space.

The Windows NT I/O subsystem provides flexible support for queue management.[18] What the J300 achieved on the UNIX and OpenVMS platforms through the command and event queues can be accomplished on the Windows NT platform using built-in support from the I/O manager. A queue of pending requests (in the form of I/O request packets) may be associated with each device. The use of I/O packets is similar to the use of command and event packets in the J300 video software.

Summary

This paper describes the design and implementation of the software architecture for the Sound & Motion J300 product, Digital's first commercially available multimedia hardware adapter that incorporates audio and video compression. The presentation focused on those aspects of the design that place special emphasis on performance, on providing an intuitive API, and on supporting a client-server model of computing. The software architecture has been successfully implemented on the OpenVMS, Microsoft Windows NT, and Digital UNIX platforms. It is the basis for Digital's recent PCI-based video adapter cards: FullVideo Supreme and FullVideo Supreme JPEG.

The goals that influenced the J300 design have largely been realized, and the software is mature. Digital is expanding upon ideas incorporated in the design. For example, one potential area for improvement is to replace the FIFO service policy in the various queues with a priority-based mechanism. A second possible improvement is to increase the usage of the hardware between periodic operations like video capture. In terms of portability, the idea of leaving device-specific programming outside the kernel driver can be expanded upon to design device-independent kernel-mode drivers, thus lowering overall development costs. Digital is actively investigating these and other such enhancements made possible by the success of the J300 project.

Acknowledgments

A number of people are responsible for the success of the J300 family of products. The author gratefully acknowledges the contributions of members of the J300 software and hardware development teams. In particular, special thanks to Bernard Szabo, the project leader for the J300 software; Paul Gauthier, for his invaluable assistance in getting the video library completed and debugged; John Hainsworth, for implementing the device-independent portion of the audio library; Davis Pan, for writing the DSP programs; and Robert Ulichney, for his guidance with the design and implementation of the video rendering subsystem. The J300 hardware design team was lead by Ken Correll and included Tim Hellman, Peter Antonios, Rudy Stalzer, and Tom Fitzpatrick. Nagi Sivananjaiah wrote the diagnostics that served us well in isolating hardware problems. Thanks also to members of the Multimedia Services Group, including Jim Ludwig, Ken Chiquoine, Leela Oblichetti, and Chip Dancy, for being instrumental in incorporating the J300, FullVideo Supreme, and FullVideo Supreme JPEG into Digital's multimedia server, and to Susan Yost, our tireless product manager, for diligently ensuring that the development teams remained on track.

References

1. *Information Technology—Digital Compression and Coding of Continuous-tone Still Images, Part 1: Requirements and Guidelines,* ISO/IEC 10918-1: 1994 (March 1994).

2. *Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbit/s—Part 2: Video,* ISO/IEC 11172-2: 1993 (1993).

3. P. Bahl, P. Gauthier, and R. Ulichney, "Software-only Compression, Rendering, and Playback of Digital Video," *Digital Technical Journal,* vol. 7, no. 4 (1995, this issue): 52–75.

4. A. Banerjea et al., "The Tenet Real-Time Protocol Suite: Design, Implementation, and Experiences," TR-94-059 (Berkeley, Calif.: International Computer Science Institute, November 1994), also in *IEEE/ACM Transactions on Networking* (1995).

5. A. Banerjea, E. Knightly, F. Templin, and H. Zhang, "Experiments with the Tenet Real-Time Protocol Suite on the Sequoia 2000 Wide Area Network," *Proceedings of the ACM Multimedia '94,* San Francisco, Calif. (1994).

6. W. Fenner, L. Berc, R. Frederick, and S. McCanne, "RTP Encapsulation of JPEG Compressed Video," Internet Engineering Task Force, Audio-Video Transport Working Group (March 1995). (Internet draft)

7. S. McCanne and V. Jacobson, "vic: A Flexible Framework for Packet Video," *Proceedings of the ACM Multimedia '95,* San Francisco, Calif. (1995).

8. M. Altenhofen et al., "The BERKOM Multimedia Collaboration Service," *Proceedings of the ACM Multimedia '93,* Anaheim, Calif. (August 1993): 457–463.

9. K. Correll and R. Ulichney, "The J300 Family of Video and Audio Adapters: Architecture and Hardware Design," *Digital Technical Journal,* vol. 7, no. 4 (1995, this issue): 20–33.

10. S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3 BSD UNIX Operating System* (Reading, Mass.: Addison-Wesley, 1989): 51–53.

11. *Pulse Code Modulation (PCM) of Voice Frequencies,* CCITT Recommendation G.711 (Geneva: International Telecommunications Union, 1972).

12. L. Rabiner and R. Schafer, *Digital Processing of Speech Signals* (Englewood Cliffs, N.J.: Prentice-Hall, 1978).

13. D. Lee and S. Sudharsanan, "Design of a Motion JPEG (M/JPEG) Adapter Card," in *Digital Video Compression on Personal Computers: Algorithms and Technology, Proceedings of SPIE,* vol. 2187, San Jose, Calif. (February 1994): 2–12.

14. M. Boliek and J. Allen, "JPEG Image Compression Hardware Implementation with Extensions for Fixed-rate and Compressed-image Editing Applications," in *Digital Video Compression on Personal Computers: Algorithms and Technology, Proceedings of SPIE,* vol. 2187, San Jose, Calif. (February 1994): 13–22.

15. J. Pasquale, "I/O System Design for Intensive Multimedia I/O," *Proceedings of the Third IEEE Workshop on Workstation Operation Systems,* Asilomar, Calif. (October 1991): 56–67.

16. K. Fall and J. Pasquale, "Improving Continuous-media Playback Performance with In-kernel Data Paths," *Proceedings of the IEEE Conference on Multimedia Computing and Systems,* Boston, Mass. (June 1994): 100–109.

17. H. Kitamura, K. Taniguchi, H. Sakamoto, and T. Nishida, "A New OS Architecture for High Performance Communication over ATM Networks," *Proceedings of the Workshop on Network and Operating System Support for Digital Audio and Video* (April 1995): 87–91.

18. *Microsoft Windows NT Device Driver Kit* (Redmond, Wash.: Microsoft Corporation, January 1994).

Biography

Paramvir Bahl
Paramvir Bahl received B.S.E.E. and M.S.E.E. degrees in 1987 and 1988 from the State University of New York at Buffalo. Since joining Digital in 1988, he has contributed to several seminal multimedia products involving both hardware and software for digital video. Recently, he led the development of software-only video compression and video rendering algorithms. A principal engineer in the Systems Business Unit, Paramvir received Digital's Doctoral Engineering Fellowship Award and is completing his Ph.D. at the University of Massachusetts. There, his research has focused on techniques for robust video communications over mobile radio networks. He is the author and coauthor of several scientific publications and a pending patent. He is an active member of the IEEE and ACM, serving on program committees of technical conferences and as a referee for their journals. Paramvir is a member of Tau Beta Pi and a past president of Eta Kappa Nu.