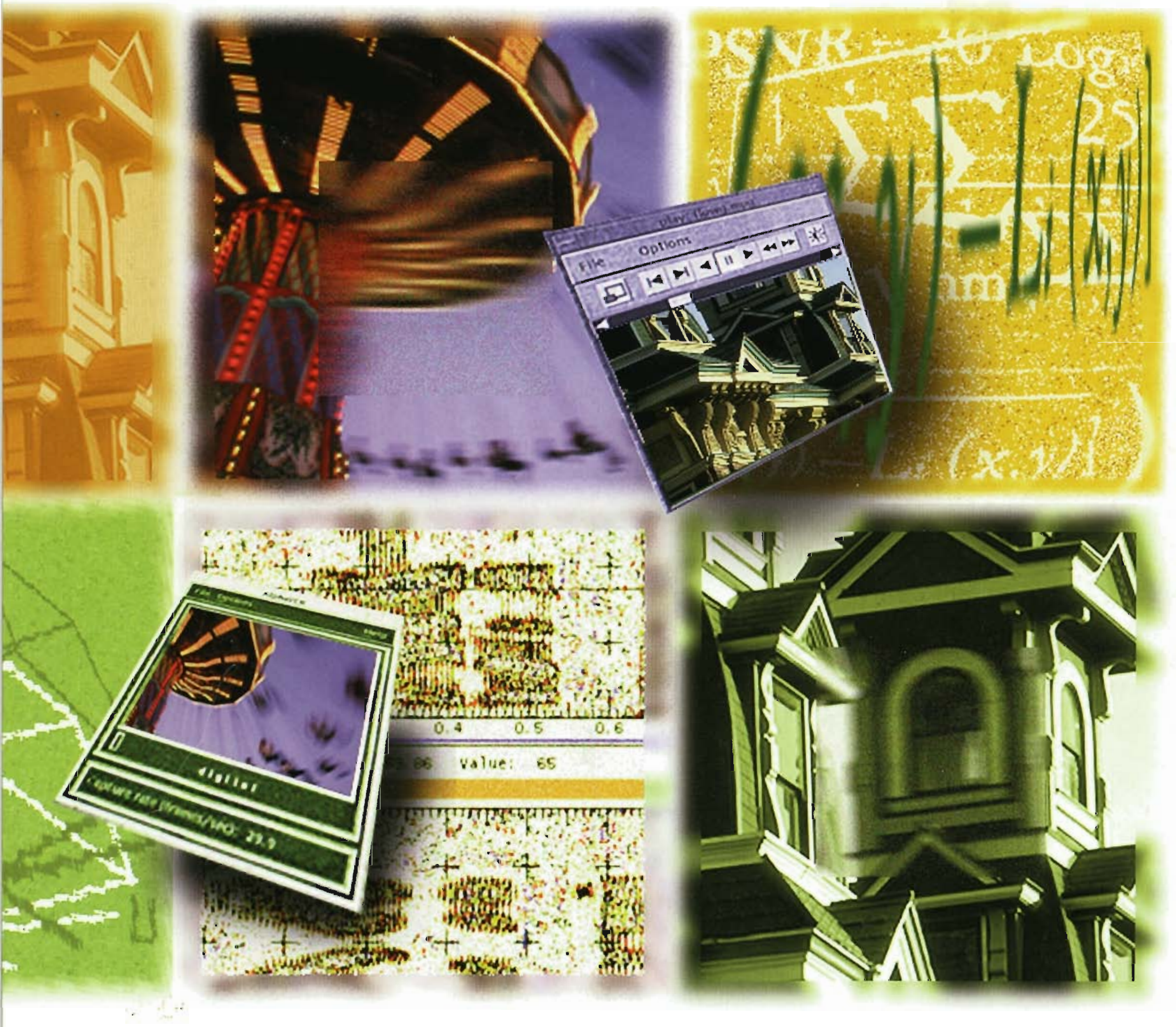
Digital Technical Journal

AUDIO AND VIDEO TECHNOLOGIES

UNIX AVAILABLE SERVERS,
REAL-TIME DEBUGGING TOOLS



Volume 7 Number 4
1995

Cover Design
The concept for the cover graphic is derived from the Video Odyssey screen saver application, which allows users to display full-motion video images on their screens in a variety of modes. The screen saver application was built to test a software architecture that organizes the functionality of video compressors and renderers into a unifying software interface. This software-only approach to digital video is one of the topics in the feature section Audio and Video Technologies in this issue.

The cover was designed by Lucinda O'Neill of Digital's Design Group. Our thanks go to authors Victor Bahl and Paul Gauthier for providing the screen saver, and to author Bill Hallahan for the DECtalk Software synthetic speech spectrogram used in the cover graphic.

Correction for Vol. 7 No. 3 Cover Design Description In describing the cover of the previous issue, vol. 7 no. 3, we neglected to properly credit the sources of the cover images. The visualizations on the front and back covers were created by Jonathan Shade using the computational resources of the San Diego Supercomputer Center. We thank Jonathan and the Center for the use of these images.

The *Digital Technical Journal* is a refereed journal published quarterly by Digital Equipment Corporation, 30 Porter Road LJO2/D10, Littleton, Massachusetts 01460. Subscriptions to the *Journal* are $40.00 (non-U.S. $60) for four issues and $75.00 (non-U.S. $115) for eight issues and must be prepaid in U.S. funds. University and college professors and Ph.D. students in the electrical engineering and computer science fields receive complimentary subscriptions upon request. Orders, inquiries, and address changes should be sent to the *Digital Technical Journal* at the published-by address. Inquiries can also be sent electronically to dtj@digital.com. Single copies and back issues are available for $16.00 each by calling DECdirect at 1-800-DIGITAL (1-800-344-4825). Recent back issues of the *Journal* are also available on the Internet at http://www.digital.com/info/DTJ/home.html. Complete Digital Internet listings can be obtained by sending an electronic mail message to info@digital.com.

Digital employees may order subscriptions through Readers Choice by entering VTX PROFILE at the system prompt.

Comments on the content of any paper are welcomed and may be sent to the managing editor at the published-by or network address.

The information in the *Journal* is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation or by the companies herein represented. Digital Equipment Corporation assumes no responsibility for any errors that may appear in the *Journal*.

ISSN 0898-901X

Documentation Number EY-U002E-TJ

Book production was done by Quantic Communications, Inc.

Contents

Editor's Introduction

This issue's opening section features audio and video technologies that exploit the power of Digital's 64-bit RISC Alpha systems. Papers describe new software and hardware designs that make practical such applications as text-to-speech conversion and full-motion video on the desktop. A second set of papers shifts the focus to the UNIX environment with discussions of high-availability services and of Encore Computer Corporation's new real-time debugging tool.

The opening paper for the audio and video section references an audio technology that physicist Stephen Hawking uses to convert the text he types to highly intelligible synthetic speech. Recently, engineers have ported this mature 10-year-old hardware technology, called DECtalk, to text-to-speech software. Bill Hallahan explains that the computational power of Digital's Alpha systems now makes it possible for a software speech synthesizer to simultaneously convert many text streams to speech without overloading a workstation. After reviewing relevant speech terminology and popular synthesis techniques, he describes DECtalk Software multithreaded processing and the new text-to-speech application programming interface for UNIX and NT workstations.

Video technologies—full-motion video on workstations—also capitalize on the high performance of Alpha systems. In the first of four papers focused on digital video, Ken Correll and Bob Ulichney present the J300 video and audio adapter architecture. To improve on past full-motion video implementations, designers sought to allow video data to be treated the same as any other data type in a workstation. The authors review the J300 features, including a versatile color-map rendering system, and the subsystem design decisions made to keep product costs low.

Victor Bahl then presents the J300 software that controls the hardware. The challenge for software designers was to obtain real-time performance from a non-real-time operating system. A description of the video subsystem highlights the video library and an innovative use of queues in achieving good performance. This software architecture has been implemented on OpenVMS, Windows NT, and Digital UNIX platforms.

A third paper on video technology looks at delivering video without specialized hardware, that is, a software-only architecture for general-purpose computers that provides access to video codecs and renderers through a flexible application programming interface. Again, faster processors make a software-only solution possible at low cost. Authors Victor Bahl, Paul Gauthier, and Bob Ulichney preface the paper with an overview of industry-standard codecs and compression schemes. They then discuss the creation of the software video library, its architecture, and its implementation of video rendering that parallels the J300 hardware.

The final paper in the audio and video technologies section explicitly raises the question of what features are best implemented in hardware and what in software. The context for the question is a graphics accelerator chip design that integrates traditional synthetic graphics features and video image display features—until now, implemented separately. Larry Seiler and Bob Ulichney describe the video processing implemented differently in two chips, both of which offer significantly higher performance with minimal additional logic.

The common theme of our second section is the UNIX operating system. Larry Cohen and John Williams present the DECsafe Available Server Environment (ASE), which provides high availability for applications running on Digital UNIX systems. They describe the ASE design for detection and dynamic reconfiguration around host, storage device, and network failures, and review key design trade-offs that favored software reliability and data integrity.

Mike Palmer and Jeff Russo then contrast Encore Computer Corporation's set of debug and analysis tools for real-time applications, called Parasight, with conventional UNIX tools. They examine the features that are critical in an effective real-time debugging tool, for example, the ability to attach to a running program and to analyze several programs simultaneously. A description follows of the Parasight product, which includes the features necessary for real-time debug and analysis in a set of graphical user interface tools.

Upcoming in our next issue are papers on a variety of topics, including Digital UNIX clusters, eXcursion for NT, and network services.

Jane Blake

Jane C. Blake
Managing Editor

Foreword

Robert A. Ulichney
Senior Consulting Engineer
Research and Advanced Development,
Cambridge Research Lab

"Can you dig it ... New York State Throughway's closed, Man. Far out, Man," announced a young Arlo Guthrie in the vernacular on the stage at Woodstock in 1969. Reading these words may evoke a mental picture of the event, but it sure is a lot more fun to hear and see Arlo deliver this message. Audio and video technology is the featured theme of this issue of the *Digital Technical Journal.*

Four years *before* Arlo's traffic report, in the year that a young Digital Equipment Corporation introduced the PDP-8, an interesting forecast was made. Gordon Moore, who was yet to co-found Intel, asserted in a little-noticed paper that the power and complexity of the silicon chip would double every year (later revised to every 18 months). This prediction has been generally accurate for 30 years and is today one of the most celebrated and remarkable "laws" of the computer industry.

While we enjoyed this exponential hardware ride, there was always some question about the ability of applications and software to keep up. If anything, the opposite is true. Software has been described as a gas that immediately fills the expanding envelope of hardware. Ever since the hardware envelope became large enough to begin to accommodate crude forms of audio and video, the pressure of the software gas has been great indeed. Digitized audio and video represent enormous amounts of data and stress the capacities of real-time processing and transmission systems.

Digital has participated in expanding the envelope and in filling it; its hardware performance is record-breaking and its audio and video technologies are state-of-the-art. Looking specifically at the four categories into which computer companies segment audio and video technologies, Digital is making contributions in each of these: analysis, synthesis, compression, and input/output.

MIT's Nicholas Negroponte believes that practical analysis, or interpretation, of digitized audio and video will be the next big advance in the computer industry, where nothing has changed in human input (keyboard and pointing device) since, well, the Woodstock era. Digital is actively investigating methods for speaker-independent speech recognition and, in the area of video analysis, means to automatically detect, track, and recognize people.

The synthesis of still and motion video, more commonly referred to as computer graphics, has traditionally been a much larger area of focus than the handling of sampled video. Synthesis of audio, or text-to-speech conversion, is the topic of one of the papers in this issue; DECtalk is largely considered to be the best such synthesis mechanism available.

When audio or video data are represented symbolically, as is the case after analysis, or prior to synthesis, a most efficient form of compression is implicitly employed. However, the task of storing or transmitting the raw digitized signal can be overwhelming, especially at high sampling rates. Compression techniques are relied upon to ease the volume of this data in two ways: (1) reducing statistical redundancy and (2) pruning data that will not be noticed by exploiting what is known about human perceptual systems. In this climate of interoperability and open systems, Digital recognizes the importance of adhering to accepted standards for audio and video compression versus the promotion of some proprietary representation.

The last category is that of I/O. Audio and video input require a means for signal acquisition and analog-to-digital conversion. The focus here is on preserving the integrity of the signal as opposed to interpreting the data. Proper rendering is needed for good-quality output, along with digital-to-analog conversion. For both audio and video, trade-offs must be made to accommodate the highest degree of sampling resolution in time and amplitude.

Digital is a leader in the area of video rendering with our AccuVideo technology, aspects of which are described in part in three papers in this issue. Video rendering incorporates all processing that is required to tailor video to a particular target display. This includes scaling and filtering, color adjustment, dithering, and color-space conversion from video's luminance-chrominance representation to RGB. In its most general form, Digital's rendering technology will optimize display quality given *any* number of available colors.

The earliest form of AccuVideo appeared in a 1989 testbed, known internally as Pictor. This led to the widely distributed research prototype called Jvideo in 1991. Jvideo was a TURBOchannel bus option with JPEG compression and decompression and was the first prototype to combine dithering with color-space conversion. Jvideo was the basis for design of the Sound & Motion J300 product, which included a remarkably improved dither method. A follow-on to J300 is a PCI-bus version called FullVideo Supreme.

In products that render RGB data instead of video, Digital's rendering technology is referred to as AccuLook; except for this one difference, the rest of the rendering pipeline is identical to AccuVideo. AccuLook products include graphics options for workstations: ZLX-E (SFB+) designed for the TURBOchannel and ZLXp-E (TGA) designed as an entry-level product for the PCI bus.

AccuVideo rendering is a key feature in the DECchip 21130 PC graphics chip and in the TGA2 high-end workstation graphics chip. While noted for its high image quality, AccuVideo is also efficiently implemented in software; it is available as part of a tool kit with every Digital UNIX, OpenVMS, and Windows NT platform.

With Moore's law on the loose, it can be argued that hardware implementations of video rendering are not justified as software-only versions grow in speed. Although today's processors can indeed handle the playback of video by both decompressing and rendering at a quarter of full size, little is left for doing anything else. Moreover, users will want to scale up the display sizes, and perhaps add multiple video streams—and still be able to use their processors to do other things. For the near term, hardware video rendering is justified.

The five papers that make up the audio and video technology theme of this issue are but a small sampling of the work under way in this area at Digital; look for more papers to follow in subsequent issues of this *Journal*. As the audio and video gas continues to fill the ever-expanding hardware envelope, we look forward to an enriched and more natural experience with computing devices. Arlo's Woodstock pals would likely agree that this sounds like more fun.

William I. Hallahan

DECtalk Software: Text-to-Speech Technology and Implementation

DECtalk is a mature text-to-speech synthesis technology that Digital has sold as a series of hardware products for more than ten years. Originally developed by Digital's Assistive Technology Group (ATG) as an alternative to a character-cell terminal and for telephony applications, today DECtalk also provides visually handicapped people access to information. DECtalk uses a digital formant synthesizer to simulate the human vocal tract. Before the advent of the Alpha processor, the computational demands of this synthesizer placed an extreme load on a workstation. DECtalk Software has an application programming interface (API) that is supported on multiple platforms and multiple operating systems. This paper describes the various text-to-speech technologies, the DECtalk Software architecture, and the API. The paper also reports our experience in porting the DECtalk code base from the previous hardware platform.

During the past ten years, advances in computer power have created opportunities for voice input and output. Many major corporations, including Digital, provide database access through the telephone. The advent of Digital's Alpha processor has changed the economics of speech synthesis. Instead of an expensive, dedicated circuit card that supports only a single channel of synthesis, system developers can use an Alpha-based workstation to support many channels simultaneously. In addition, since text-to-speech conversion is a light load for an Alpha processor, application developers can freely integrate text to speech into their products.

Digital's DECtalk Software provides natural-sounding, highly intelligible text-to-speech synthesis. It is available for the Digital UNIX operating system on Digital's Alpha-based platforms and for Microsoft's Windows NT operating system on both Alpha and Intel processors. DECtalk Software provides an easy-to-use application programming interface (API) that is fully integrated with the computer's audio subsystem. The text-to-speech code was ported from the software for the DECtalk PC card, a hardware product made by Digital's Assistive Technology Group. This software constitutes over 30 man years of development effort and contains approximately 160,000 lines of C programming language code.

This paper begins by discussing the features of DECtalk Software and briefly describing the various text-to-speech technologies. It then presents a description of the DECtalk Software architecture and the API. Finally, the paper relates our experience in porting the DECtalk code base.

Features of DECtalk Software

The DECtalk Software development kit consists of a shared library (a dynamic link library on Windows NT), a link library, a header file that defines the symbols and functions used by DECtalk Software, sample applications, and sample source code that demonstrates the API.

DECtalk Software supports nine preprogrammed voices: four male, four female, and one child's voice. Both the API and in-line text commands can control the voice, the speaking rate, and the audio volume. The volume command supports stereo by providing independent control of the left and right channels. Other in-line commands play wave audio files, generate single tones, or generate dual-tone multiple-frequency (DTMF) signals for telephony applications.

Using the text-to-speech API, applications can play speech through the computer's audio system, write the speech samples to a wave audio file, or write the speech samples to buffers supplied by the application. DECtalk Software produces speech in 3 audio formats: 16-bit pulse code modulation (PCM) samples at an 11,025-hertz (Hz) sample rate, 8-bit PCM samples at an 11,025-Hz sample rate, and μ-law encoded 8-bit samples at an 8,000-Hz sample rate. The first two formats are standard multimedia audio formats for personal computers (PCs). The last format is the standard encoding and rate used for telephony applications.

The API can also load a user-generated dictionary that defines the pronunciation of application-specific words. The development kit provides a window-based tool to generate these dictionaries. The kit also contains a window-based application to speak text and an electronic mail-notification program. Sample source code includes a simple window-based application that speaks text, a command line application to speak text, and a speech-to-memory sample program.

The version of DECtalk Software for Windows NT also provides a text-to-speech dynamic data exchange (DDE) server. This server integrates with other applications such as Microsoft Word. Users can select text in a Word document and then proofread the text merely by clicking a button. This paper was proofread using DECtalk Software running a native version of Microsoft Word on an AlphaStation workstation.

Speech Terms and DECtalk Software

Human speech is produced by the vocal cords in the larynx, the trachea, the nasal cavity, the oral cavity, the tongue, and the lips. Figure 1 shows the human speech organs. The glottis is the space between the vocal cords. For voiced sounds such as vowels, the vocal cords produce a series of pulses of air. The pulse repetition frequency is called the glottal pitch. The pulse train is referred to as the glottal waveform. The rest of the articulatory organs filter this waveform.[1] The trachea, in conjunction with the oral cavity, the tongue, and the lips, acts like a cascade of resonant tubes of varying widths. The pulse energy reflects backward and forward in these organs, which causes energy to propagate best at certain frequencies. These are called the formant frequencies.

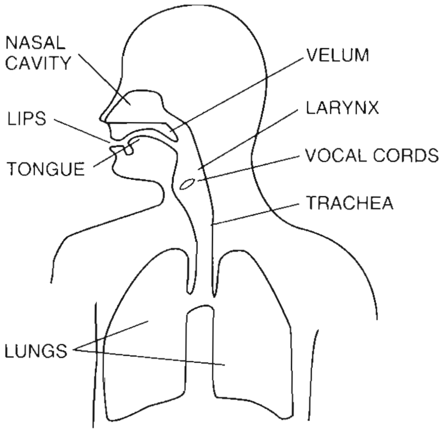


Figure 1
The Speech Organs

The primary discrimination cues for different vowel sounds are the values of the first and second formant frequency. Vowels are either front, mid, or back vowels, depending on the place of articulation. They are either rounded or unrounded, depending on the position of the lips. American English has 12 vowel sounds. Diphthongs are sounds that change smoothly from one vowel to another, such as in *boy, bow,* and *bay.* Other voiced sounds include the nasals *m, n,* and *ng* (as in *ing*). To produce nasals, a person opens the velar flap, which connects the throat to the nasal cavity. Liquids are the vowel-like sounds *l* and *r.* Glides are the sounds *y* (as in *you*) and *w* (as in *we*).

Breath passing through a constriction creates turbulence and produces unvoiced sounds. *f* and *s* are unvoiced sounds called fricatives. A stop (also called a plosive) is a momentary blocking of the breath stream followed by a sudden release. The consonants *p, b, t, d, k,* and *g* are stop consonants. Opening the mouth and exhaling rapidly produces the consonant *h.* The *h* sound is called an aspirate. Other consonants such as *p, t,* and *k* frequently end in aspiration, especially when they start a word. An affricative is a stop immediately followed by a fricative. The English sounds *ch* (as in *chew* and *j* (as in *jar*) are affricates.

These sounds are all American English phonemes. Phonemes are the smallest units of speech that distinguish one utterance from another in a particular language.[2] An allophone is an acoustic manifestation of a phoneme. A particular phoneme may have many allophones, but each allophone (in context) will sound like the same phoneme to a speaker of the language that defines the phoneme. Another way of saying this is, if two sounds have different acoustic manifestations, but the use of either one does not change the meaning of an utterance, then by definition, they are the same phoneme.

Phones are the sets of all phonemes and allophones for all languages. Linguists have developed an international phonetic alphabet (IPA) that has symbols for almost all phones. This alphabet uses many Greek letters that are difficult to represent on a computer. American linguists have developed the Arpabet phoneme alphabet to represent American English phonemes using normal ASCII characters. DECtalk Software supports both the IPA symbols for American English and the Arpabet alphabet. Extra symbols are provided that either combine certain phonemes or specify certain allophones to allow the control of fine speech features. Table 1 gives the DECtalk Software phonemic symbols.

Speech researchers often use the short-term spectrum to represent the acoustic manifestation of a sound. The short-term spectrum is a measure of the frequency content of a windowed (time-limited) portion of a signal. For speech, the time window is typically between 5 milliseconds and 25 milliseconds, and the pitch frequency of voiced sounds varies from 80 Hz to 280 Hz. As a result, the time window ranges from slightly less than one pitch period to several pitch periods. The glottal pitch frequency changes very little in this interval. The other articulatory organs move so little over this time that their filtering effects do not change appreciably. A speech signal is said to be stationary over this interval.

Table 1
DECtalk Software Phonemic Symbols

Consonants		Vowels and Diphthongs	
b	*b*et	aa	B*o*b
ch	*ch*in	ae	b*a*t
d	*d*ebt	ah	b*u*t
dh	*th*is	ao	b*ou*ght
el	bott*le*	aw	b*ou*t
en	butt*on*	ax	*a*bout
f	*f*in	ay	b*i*te
g	*g*uess	eh	b*e*
hx	*h*ead	ey	b*a*ke
jh	*g*in	ih	b*i*t
k	*K*en	ix	kiss*e*s
l	*l*et	iy	b*ea*t
m	*m*et	ow	b*oa*t
n	*n*et	oy	b*oy*
nx	si*ng*	rr	b*ir*d
p	*p*et	uh	b*oo*k
r	*r*ed	uw	l*u*te
s	*s*it	yu	c*u*te
sh	*sh*in	**Allophones**	
t	*t*est	dx	ri*d*er
th	*th*in	lx	e*l*ectric
v	*v*est	q	we *e*at
w	*w*et	rx	o*r*ation
yx	*y*et	tx	La*t*in
z	*z*oo	**Silence**	
zh	a*z*ure	_	(underscore)

The spectrum has two components for each frequency measured, a magnitude and a phase shift. Empirical tests show that sounds that have identical spectral magnitudes sound similar. The relative phase of the individual frequency components plays a lesser role in perception. Typically, we perceive phase differences only at the start of low frequencies and only occasionally at the end of a sound. Matching the spectral magnitude of a synthesized phoneme (allophone) with the spectral magnitude of the desired phoneme (taken from human speech recordings) always improves intelligibility.[3] This is the synthesizer calibration technique used for DECtalk Software.

A spectrogram is a plot of spectral magnitude slices, with frequency on the *y* axis and time on the *x* axis. The spectral magnitudes are specified either by color or by saturation for two-color plots. Depending on the time interval of the spectrum window, either the pitch frequency harmonics or the formant structure of speech may be viewed. It is even possible to ascertain what is said from a spectrogram. Figure 2 shows spectrograms of both synthetic and human speech for the same phrase. The formant frequencies are the dark regions that move up and down as the speech organs change position. Fricatives and aspiration are characterized by the presence of high frequencies and usually have much less energy than the formants.

The bandwidth of speech signals extends to over 10 kilohertz (kHz) although most of the energy is confined below 1,500 Hz. The minimum intelligible bandwidth for speech is about 3 kHz, but using this bandwidth, the quality is poor. A telephone's bandwidth is 3.2 kHz. The DECtalk PC product has a speech bandwidth just under 5 kHz, which is the same as the audio bandwidth of an AM broadcast station. The sample rate of a digital speech system must be at least twice the signal bandwidth (and might have to be higher if the signal is a bandpass signal), so the DECtalk PC uses a 10-kHz sample rate. This bandwidth represents a trade-off between speech quality and the amount of calculation (or CPU loading). The DECtalk Software synthesizer rate is 11,025 Hz, which is a standard PC sample rate. An 8-kHz rate is provided to support telephony applications.

People often perceive acoustic events that have different short-term spectral magnitudes as the same phoneme. For example, the *k* sound in the words *kill*

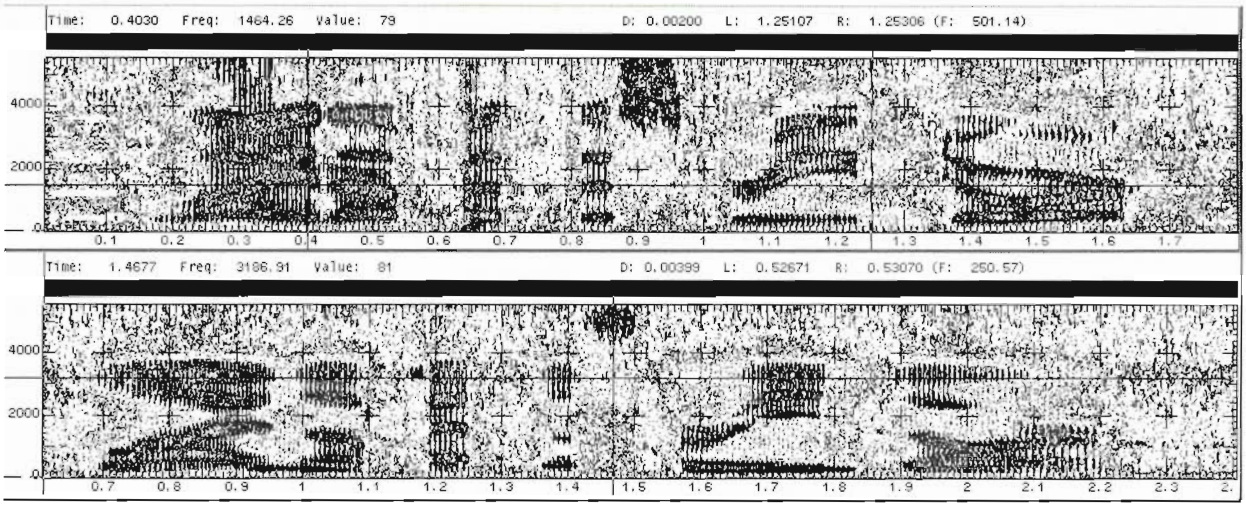


Figure 2
Two Spectrograms of the Utterance "Line up at the screen door." The upper spectrogram is the author's speech. The lower spectrogram is synthetic speech produced by DECtalk Software.

and *cool* have very different magnitude spectra. An American perceives the two spectra as the same sound; however, the sounds are very different to someone from Saudi Arabia. A Japanese person does not perceive any difference between the words *car* and *call.* To an English speaker, the *r* and the *l* sound different even though they have nearly identical magnitude spectra. The *l* sounds in the words *call* and *leaf* are different spectrally (acoustically) but have the same sound. Thus they are the same phoneme in English.

Several allophones are required to represent the *k* phoneme. Most consonant phonemes require several different allophones because the vowel sounds next to them change their acoustic manifestations. This effect, called coarticulation, occurs because it is often unnecessary for the articulatory organs to reach the final position used to generate a phoneme; they merely need to gesture toward the final position. Another type of coarticulation is part of the grammar of a language. For example, the phrase *don't you* is often pronounced *doan choo.*

All allophones that represent the phoneme *k* are produced by closing the velum and then suddenly opening it and releasing the breath stream. Speakers of the English language perceive all these allophones as the same sound, which suggests that synthesis may be modeled by an articulatory model of speech production. This model would presumably handle coarticulation effects that are not due to grammar. It is currently not known how to consistently determine speech organ positions (or control strategies) directly from acoustic speech data, so articulatory models have had little success for text-to-speech synthesis.[4]

For English, the voicing pitch provides cues to clause boundaries and meaning. Changing the frequency of the vibration of the vocal cords varies the pitch. Intonation is the shape of the pitch variation across a clause. The sentence "Tim is leaving." is pronounced differently than "Tim is leaving?" The latter form requires different intonation, depending on whether the intent is to emphasize that it is "Tim" who is leaving, or that "leaving" is what Tim is to do. A word or phrase is stressed by increasing its pitch, amplitude, or duration, or some combination of these. Intonation includes pitch changes due to stress and normal pitch variation across a clause. Correct intonation is not always possible because it requires speech understanding. DECtalk Software performs an analysis of clause structure that includes the form classes of both words and punctuation and then applies a pitch contour to a clause. The form class definitions include symbols for the parts of speech (article, adjective, adverb, conjunction, noun, preposition, verb, etc.) and symbols to indicate if the word is a number, an abbreviation, a homograph, or a special word (requiring special proprietary processing). For the sentence, "Tim is leaving?" the question mark causes DECtalk Software to raise the final pitch, but no stress is put on "Tim" or "leaving." Neutral intonation sometimes sounds boring, but at least it does not sound foolish.

Text-to-Speech Synthesis Techniques

Early attempts at text-to-speech synthesis assembled clauses by concatenating recorded words. This technique produces extremely unnatural-sounding speech.

In continuous speech, word durations are often shortened and coarticulation effects can occur between adjacent words. There is also no way to adjust the intonation of recorded words. A huge word database is required, and words that are not in the database cannot be pronounced. The resulting speech sounds choppy.

Another word concatenation technique uses recordings of the formant patterns of words. A formant synthesizer smoothes formant transitions at the word boundaries. A variation of this technique uses linear predictive coded (LPC) words. An advantage of the formant synthesizer is that the pitch and duration of words may be varied. Unfortunately, since the phoneme boundaries within a word are difficult to determine, the pitch and duration of the individual phonemes cannot be changed. This technique also requires a large database. Again, a word can be spoken only if it is in the database. In general, the quality is poor, although this technique has been used with some success to speak numbers.

A popular technique today is to store actual speech segments that contain phonemes and phoneme pairs. These speech segments, known as diphones, are obtained from recordings of human speech. They contain all coarticulation effects that occur for a particular language. Diphones are concatenated to produce words and sentences. This solves the coarticulation problem, but it is impossible to accurately modify the pitch of any segment. The intonation across a clause is generally incorrect. Even worse, the pitch varies from segment to segment within a word. The resulting speech sounds unnatural, unless the system is speaking a phrase that the diphones came from (this is a devious marketing ploy). Nevertheless, diphone synthesis produces speech that is fairly intelligible. Diphone synthesis requires relatively little compute power, but it is memory intensive. American English requires approximately 1,500 diphones; diphone synthesis would have to provide a large database of approximately 3 megabytes for each voice included by the system.

DECtalk Software uses a digital formant synthesizer. The synthesizer input is derived from phonemic symbols instead of stored formant patterns as in a conventional formant synthesizer. Intonation is based on clause structure. Phonetic rules determine coarticulation effects. The synthesizer requires only two tables, one for each gender, to map allophonic variations of each phoneme to acoustic events. Modification of vocal tract parameters in the synthesizer allows the system to generate multiple voices without a significant increase in storage requirements. (The DECtalk code and data occupy less than 1.5 megabytes.)

Poor-quality speech is difficult to understand and causes fatigue. Linguists use standard phoneme recognition tests and comprehension tests to measure the intelligibility of synthetic speech. The DECtalk family of products achieves the highest test scores of all text-to-speech systems on the market.[5] Visually handicapped individuals prefer DECtalk over all other text-to-speech systems.

How DECtalk Software Works

DECtalk Software consists of eight processing threads: (1) the text-queuing thread, (2) the command parser, (3) the letter-to-sound converter, (4) the phonetic and prosodic processor, (5) the vocal tract model (VTM) thread, (6) the audio thread, (7) the synchronization thread, and (8) the timer thread. The text, VTM, audio, synchronization, and timer threads are not part of the DECtalk PC software (the DECtalk PC VTM is on a special Digital Signal Processor) and have been added to DECtalk Software. The audio thread creates the timer thread when the text-to-speech system is initialized. Since the audio thread does not usually open the audio device until a sufficient number of audio samples are queued, the timer thread serves to force the audio to play in case any samples have been in the queue too long. The DECtalk Software threads perform serial processing of data as shown in Figure 3.

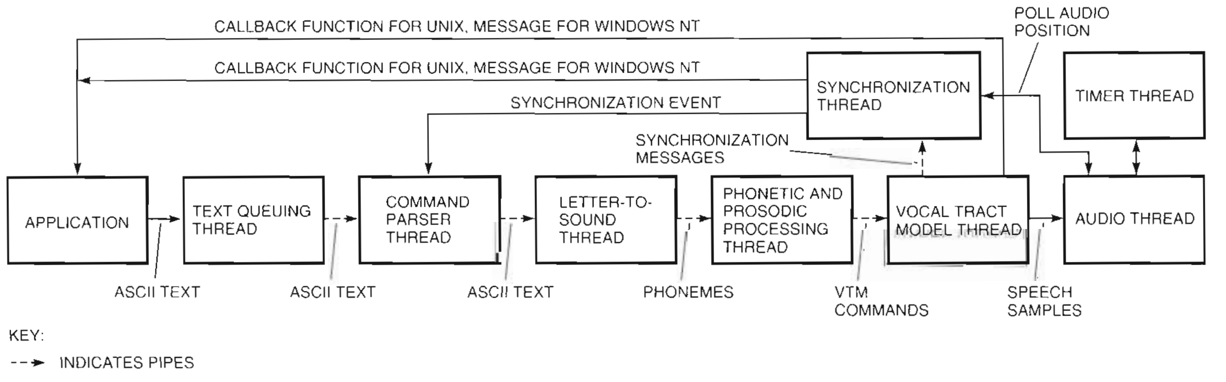


Figure 3
The DECtalk Software Architecture for Windows NT

Multithreading allows a simple and efficient means of throttling data in multistage, real-time systems. Each thread passes its output to the next thread through pipes. Each thread has access to two pipe handles, one for input and one for output. Most threads consist of a main loop that has one or more calls to a read_pipe function followed by one or more calls to a write_pipe function. The write_pipe function will block processing and suspend the thread if the specified pipe does not have enough free space to receive the specified amount of data. The read_pipe function will block processing and suspend the thread if the specified pipe does not contain the requested amount of data. Thus an active thread will eventually become idle, either because there is not enough input data, or because there is no place to store its output.

The pipes are implemented as ring buffers. The ring buffer item count is protected by mutual-exclusion objects on the Digital UNIX operating system and by critical sections on the Windows NT operating system. The pipes are created at text-to-speech initialization and destroyed during shutdown. The DECtalk Software team implemented these pipes because the pipe calls supplied with the Digital UNIX and Windows NT operating systems are for interprocess communication and are not as efficient as our pipes.

The DECtalk Software threads all used different amounts of CPU time. The data bandwidth increases at the output of every thread between the command thread and the VTM thread. Since the VTM produces audio samples at a rate exceeding 11,025 samples per second, it is no surprise that the VTM uses the most CPU time of all threads. Table 2 gives the percentage of the total application time used by each thread when the Windows NT sample application "say" is continuously speaking a large text file on an Alpha AXP 150 PC product. The output sample rate is 11,025 Hz. Note that the "say" program main thread blocks and uses virtually no CPU time after queuing the text block. These percentages have been calculated from times obtained using the Windows NT performance monitor tool.

Table 2
DECtalk Software Thread Loading

Thread	Percentage of Total Application CPU Time
Application (say.exe)	1.0
Text queue	0.2
Command parser	1.4
Letter-to-sound processing	2.4
Prosodic and phonetic processing	18.3
Vocal tract model	71.9
Audio	2.9
Synchronization	0.0
Timer	0.0
System	1.9

Because the data bandwidth increases at the output of successive threads, it is desirable to adjust the size of each of the pipes ring buffers. If one imagines that all the pipes had an infinite length (and the audio queue was infinite) and that the operating system switched thread context only when the active thread yielded, then the text thread would process all the ASCII text data before the letter-to-sound thread would run. Likewise, each successive thread would run to completion before the next thread became active. The system latency would be very high, but the thread switching would be minimized. The system would use 100 percent of the CPU until all the text was converted to audio, and then the CPU usage would become very low as the audio played out at a fixed rate. Alternatively, if all the pipes are made very short, the system latency is low. In this case, all but one of the threads will become blocked in a very short time and the startup transient in the CPU loading will be minimized. Unfortunately, the threads will constantly switch, resulting in poor efficiency. What is needed is a trade-off between these two extremes.

For a specified latency, the optimum pipe sizes that minimize memory usage for a given efficiency are in a ratio such that each pipe contains the same temporal amount of data. For example, let us assume that 64 text characters (requiring 64 bytes) are in the command thread. They produce approximately 100 phonemes (requiring 1,600 bytes) at the output of the letter-to-sound thread and approximately 750 VTM control commands (requiring 15,000 bytes) at the output of the prosodic and phonetics thread. In such a case, the size of the input pipes for the command, letter-to-sound, and prosodic and phonetic threads could be made 64, 1,600, and 15,000 bytes, respectively, to minimize pipe memory usage for the specified latency. (All numbers are hypothetical.) The pipe sizes in DECtalk Software actually increase at a slightly faster rate than necessary. We chose the faster rate because memory usage is not critical since all the pipes are small relative to other data structures. The size of the VTM input pipe is the most critical: it is the largest pipe because it supports the largest data bandwidth.

The Text Thread

The text thread's only purpose is to buffer text so the application is not blocked during text processing. An application using text-to-speech services calls the TextToSpeechSpeak API function to queue a nullterminated text string to the system. This API function copies the text to a buffer and passes the buffer (using a special message structure) to the text thread. This is done using the operating system's PostMessage function for Windows NT and a thread-safe linked list for Digital UNIX. After the text thread pipes the entire text stream to the command thread, it frees the text buffer and the message structure.

The Command Processing Thread

The command processing thread parses in-line text commands. These commands control the text-to-speech system voice selection, speaking rate, and audio volume, and adjust many other system state parameters. For DECtalk, most of these commands are of the form [: command <parameters>]. The string "[:" specifies that a command string follows. The string "]" ends a command. The following string illustrates several in-line commands.

```
[:nb][:ra 200] My name is Betty.
[:play audio.wav]
[:dial 555-1212][:tone 700 1,000]
```

This text will select the speaker voice for "Betty," select a speaking rate of 200 words per minute, speak the text "My name is Betty." and then play a wave audio file named "audio.wav." Finally, the DTMF tones for the number 555-1212 are played followed by a 700-Hz tone for 1,000 milliseconds.

Because the text-to-speech system may be speaking while simultaneously processing text in the command thread, it is necessary to synchronize the command processing with the audio. The DECtalk PC product (from which we ported the code) did not perform synchronization unless the application placed a special string before the volume command. For DECtalk Software, asynchronous control of all functions provided by the in-line commands is already available through the text-to-speech API calls. For this reason, the DECtalk Software in-line commands are all synchronous.

The DECtalk command [:volume set 70] will set the audio volume level to 70. Synchronization is performed by inserting a synchronization symbol in the text stream. This symbol is passed through the system until it reaches the VTM thread. When the VTM thread receives a synchronization symbol, it pipes a message to the synchronization thread. This message causes the synchronization thread to signal an event as soon as all audio (that was queued before the message) has been played. The volume control code in the command thread is blocked until this event is signaled. The synchronization thread also handles commands of the form [:index mark 17]. Index mark commands may be used to send a message value (in this case 17) back to an application when the text up to the index mark command has been spoken.

The command thread passes control messages such as voice selection and speaking rate to the letter-to-sound and the prosodic and phonetic processing threads, respectively. Tone commands, index mark commands, and synchronization symbols are formatted into messages and passed to the letter-to-sound thread. The command thread also pipes the input text string, with the bracketed command strings removed, to the letter-to-sound thread.

The Letter-to-Sound Thread

The letter-to-sound (LTS) thread converts ASCII text sequences to phoneme sequences. This is done using a rule-based system and a dictionary for exceptions. It is the single most complicated piece of code in all of DECtalk Software. Pronunciation of English language words is complex. Consider the different pronunciations of the string *ough* in the words *rough, through, bough, thought, dough, cough,* and *biccough*.[6] Even though the LTS thread has more than 1,500 pronunciation rules, it requires an exception dictionary with over 15,000 words.

Each phoneme is actually represented by a structure that contains a phonemic symbol and phonemic attributes that include duration, stress, and other proprietary tags that control phoneme synthesis. This is how allophonic variations of a phoneme are handled. In the descriptions that follow, the term phoneme refers either to this structure or to the particular phone specified by the phonemic symbol in this structure.

The LTS thread first separates the text stream into clauses. Clause separation occurs in speech both to encapsulate a thought and because of our limited lung capacity. Speech run together with no breaks causes the listener (and the speaker) to become fatigued. Correct clause separation is important to achieve natural intonation. Clauses are delineated by commas, periods, exclamation marks, question marks, and special words. Clause separation requires simultaneous analysis of the text stream. For example, an abbreviated word does not end a clause even though the abbreviation ends in a period. If the text stream is sufficiently long and no clause delimiter is encountered, an artificial clause boundary is inserted into the text stream.

After clause separation, the LTS thread performs text normalization. For this, the LTS thread provides special processing rules for numbers, monetary amounts, abbreviations, times, in-line phonemic sequences, and even proper names. Text normalization usually refers to text replacement, but in many cases the LTS thread actually inserts the desired phoneme sequence directly into its output phoneme stream instead of replacing the text.

The LTS thread converts the remaining unprocessed words to phonemes by using either the exception dictionary or a rule-based "morph" lexicon. (The term *morph* is derived from morpheme, the minimum unit of meaning for a language.) By combining these two approaches, memory utilization is minimized. A user-definable dictionary may also be loaded to define application-specific terms. During this conversion, the LTS thread assigns one or more form classes to each word. As mentioned previously, form class definitions include symbols for abbreviations and homographs. A homograph is a word that has more than one pronunciation, such as *alternate* or *console*. DECtalk Software pronounces most abbreviations and homographs correctly in context. An alternate pronunciation of a homograph may be forced by inserting the in-line command [:pron alt] in front of the word. DECtalk Software speaks the phrase "Dr. Smith lives on Smith Dr." correctly, as "Doctor Smith lives on Smith Drive." It uses the correct pronunciation of the homograph *lives*.

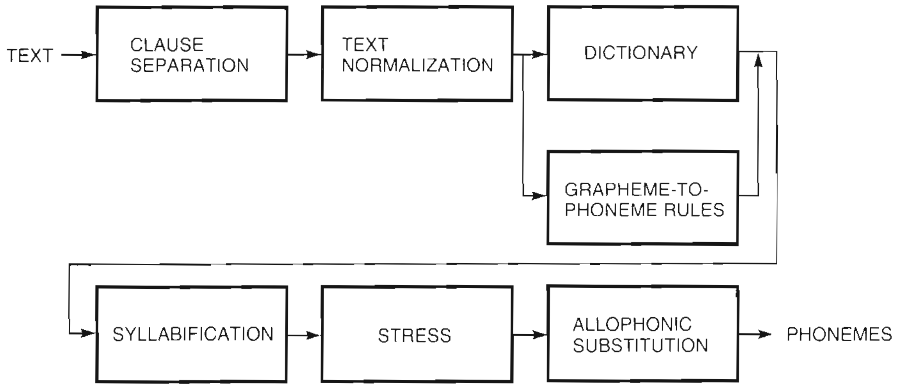
Before applying rules, the LTS thread performs a dictionary lookup for each unprocessed word in a clause. If the lookup is successful, the word's form classes and a stored phoneme sequence are extracted from the dictionary. Otherwise, the word is tested for an English suffix, using a suffix table. If a suffix is found, sometimes the form class of the word can be inferred. Suffix rules are applied, and the dictionary lookup is repeated with the new suffix-stripped word. For example, the word *testing* requires the rule, locate the suffix *ing* and remove it; whereas the word *analyzing* requires the rule, locate the suffix *ing* and replace it with *e*. The suffix rules and the dictionary lookup are recursive to handle words that end in multiple suffixes such as *endlessly*.

If the word is not in the dictionary, the LTS thread performs a decomposition of the word using morphs. DECtalk uses a morph table to look up the phonemic representation of portions of words. A morph always maps onto one or more English words and can be represented by a letter string. Morphs generally consist of one or more roots that may contain affixes and suffixes. Although new words may frequently be added to a language, new morphs are rarely added. They are essentially sound groupings that make up many of the words of a language. DECtalk contains a table with hundreds of morphs and their phonemic representations. Either a single character or a set of characters that results in a single phoneme is referred to as a grapheme. Thus this portion of the letter-to-sound conversion is referred to as the grapheme-to-phoneme translator. Figure 4 shows the architecture of the LTS thread.

Morphemes are abstract grammatical units and were originally defined to describe words that can be segmented, such as *tall*, *taller*, and *tallest*. The word *tallest* is made from the morphemes *tall* and *est*. The word *went* decomposes into the morphemes *go* and PAST. Thus a morpheme does not necessarily map directly onto a derived word. Many of the pronunciation rules are based on the morphemic representations of words.

Many morphs have multiple phonemic representations that can depend on either word or phonemic context. The correct phonemic symbols are determined by morphophonemic rules. For example, plural words that end in the morpheme *s* are spoken by appending either the *s*, the *z*, or the *eh z* plural morphemes (expressed as Arpabet phonemic symbols) at the end of the word.[7] Which allomorph is used depends on the final phoneme of the word. Allomorphs are morphemes with alternate phonetic forms. For another example requiring a morphophonemic rule, consider the final phoneme of the word *the* when pronouncing "the apple," and "the boy."

After applying many morphophonemic rules to the phonemes, the LTS thread performs syllabification, applies stress to certain syllables, and performs allophonic recoding of the phoneme stream. The LTS



Note that the grapheme-to-phoneme rules are used only if the dictionary lookup fails.

Figure 4
Block Diagram of the Letter-to-Sound Processing Thread

thread groups phonemes into syllables, using tables of legal phoneme clusters and special rules. The syllabification must be accurate, because the LTS thread applies stress between syllable boundaries.

The LTS thread then assigns either primary stress, secondary stress, or no stress to each syllable. The stress rules are applied in order. They assign stress only to syllables that have not had stress previously assigned. These rules take into account the number of syllables in a word and the positions of affixes that were found during morph decomposition of a word.

Allophonic rules are the last rules the LTS thread applies to the phoneme stream. These are really phonetic rules. Most allophonic rules are described as follows: "if phoneme A is followed by phoneme B, then modify (or delete) phoneme A (or B)." Most allophonic rules are not applied across morpheme boundaries. These rules handle many specific cases; for example, the *p* in the word *spit* is aspirated, whereas the *p* in the word *pit* is not. The *s* phoneme modifies the articulation of the *p*. The *s* phoneme is different in the words *stop* and *street* because the *r* sound is anticipated and modifies the *s* in the word *street*. This last example is called distant assimilation.

The LTS thread passes the phonemes that include durations and lexical information to the prosodic and phonetic processing thread. Tone, dial, index mark, and synchronization messages are passed unmodified through the LTS thread.

The Phonetic and Prosodic Processing Thread

The phonetic and prosodic processing (PH) thread, shown in Figure 5, converts the phoneme stream to a series of vocal tract control commands. Both prosodic rules and additional phonetic rules are applied to the input phoneme stream.[8] Prosody refers to clause-based stress, intonation, and voice quality in speech. Words are stressed to add meaning to a clause. Stress is achieved by increasing one or more of either the pitch, the duration, or the amplitude of an utterance. The phonetic rules handle coarticulation effects and adjust phoneme durations based on the form class, the clause position, and the speaking rate. One example is a rule that increases the duration of the final stressed phoneme in a clause. Additional context-dependent phonetic coarticulation rules can adjust the durations of phonemes or delete them.

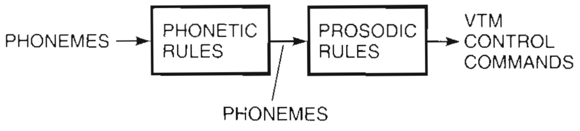


Figure 5
The Phonetic and Prosodic Processing Thread

The correct application of stress, like intonation, requires understanding, so DECtalk Software generally applies syllabic stress only as part of an intonation contour across a clause. Intonation contours are generated by fixed rules. In most clauses, the pitch rises at the start of the clause and falls at the end of the clause. This basic form is changed for questions, prepositional phrases, exclamations, compound nouns, and numbers. This intonation is also changed based on the syllabic stress assigned by the LTS thread. The PH thread can also process pitch control symbols that are placed in-line with text. These pitch commands are parsed in the command thread and pass through the LTS thread.

The PH thread uses each phoneme symbol and its context to generate any allophonic variation of the phoneme. The resulting allophone symbol indexes into one of two tables, one table for each gender. Each allophone symbol indexes a set of parameters that includes voicing source amplitude, noise source amplitude, formant frequencies, and formant bandwidths. These, along with voicing source pitch and a number of fixed speaker-dependent parameters, make up the VTM parameters. A new set of parameters is generated for every 6.4 milliseconds of speech. The VTM thread uses these parameters, which are collectively called a voice packet, to generate the speech waveform.

In addition to sending voice packets to the VTM thread, the PH thread can send a speaker packet to select a new speaking voice. The voice is selected either by an in-line text command or by the application calling a specific API function. The PH thread has fixed tables of parameters for each voice. There are many voice parameters, but some of the more interesting ones include the gender, the average pitch, the pitch range, the assertiveness, the breathiness, and the formant scale factor. The gender is used by some of the PH rules and by the PH thread to select the table used to generate voice packets. The average pitch and the pitch range are used by the PH thread to set the pitch characteristics for the VTM's voicing source. The assertiveness parameter sets the rate of fall of the pitch at the end of a clause. A high assertiveness factor results in an emphatic voice. The breathiness parameter sets the amount of noise that is mixed with the voiced path signal. The formant scale factor effectively scales the size of the speaker's trachea.

Tone, dial, index mark, and synchronization messages are passed unmodified through the PH thread.

The Vocal Tract Model Thread

The Vocal Tract Model (VTM) thread processes speaker packets, voice packets, tone messages, and synchronization messages. Speaker packets set the speaker-voice-dependent parameters of the VTM.

One of these, the formant scale factor, is multiplied by the first, second, and third formant frequencies in each voice packet. Other parameters include the values for the frequencies and bandwidths of the fourth and fifth formants, the gains for the voiced path of the VTM, the frication gain for the unvoiced path of the VTM, the speaker breathiness gain, and the speaker aspiration gain.

Each voice packet produces one speech frame of data. The output sample rate for DECtalk Software is either 8,000 Hz or 11,025 Hz. For each of these sample rates, a frame is 51 and 71 samples respectively. Each voice packet includes frequencies and bandwidths for the first, second, and third formants, the nasal antiresonator frequency, the voicing source gain, and gains for each of the parallel resonators. Figure 6 shows the basic architecture of the VTM.[9] The VTM, in conjunction with the PH rules, simulates the speech organs.

The VTM consists of two major paths, a voiced path and an unvoiced path. The voiced path is excited by a pulse generator that simulates the vocal cords. A number of resonant filters in series simulate the trachea. These cascaded resonators simulate a cascade of tubes of varying widths.[10] A nasal filter in series with the resonant tube model simulates the dominant resonance and antiresonance of the nasal cavity.[11] The cascade resonators and the nasal filter complete the "voiced" path of the VTM.

Unvoiced sounds occur as a result of chaotic turbulence produced when breath passes through a constriction. This turbulence is difficult to model. In our approach, the VTM matches the spectral magnitude of filtered noise with the spectral magnitude of the desired unvoiced phoneme (allophone). The noise source is realized by filtering the output of a uniform-distribution random number generator. Unvoiced sounds contain both resonances and antiresonances.

Another approach to obtain an appropriate frequency characteristic is to filter the noise source signal using a series of parallel resonators. A consequence of putting resonators in parallel is to create antiresonances. The positions of these antiresonances are dependent on the parallel formant frequencies, but it has been empirically determined that this model provides more than enough degrees of freedom to closely match the spectral magnitude of any unvoiced sound. The noise source generates fricatives, such as *s*, plosives, such as *p*, and aspirates, such as *h*. The noise source also contributes to some voiced sounds, such as *b*, *g*, and *z*. The noise source output may also be added to the input of the voiced path to produce aspiration. To generate breathy vowels, the parallel formant frequencies are set equal to the cascade formant frequencies.[12]

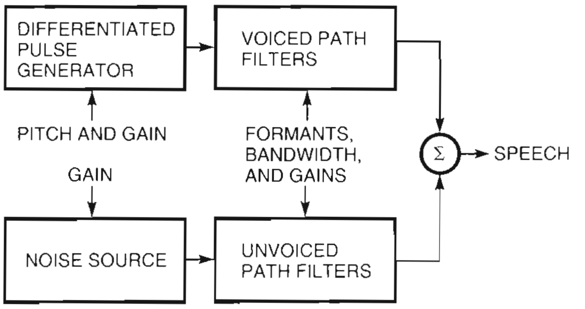


Figure 6
Basic Architecture of the Vocal Tract Model

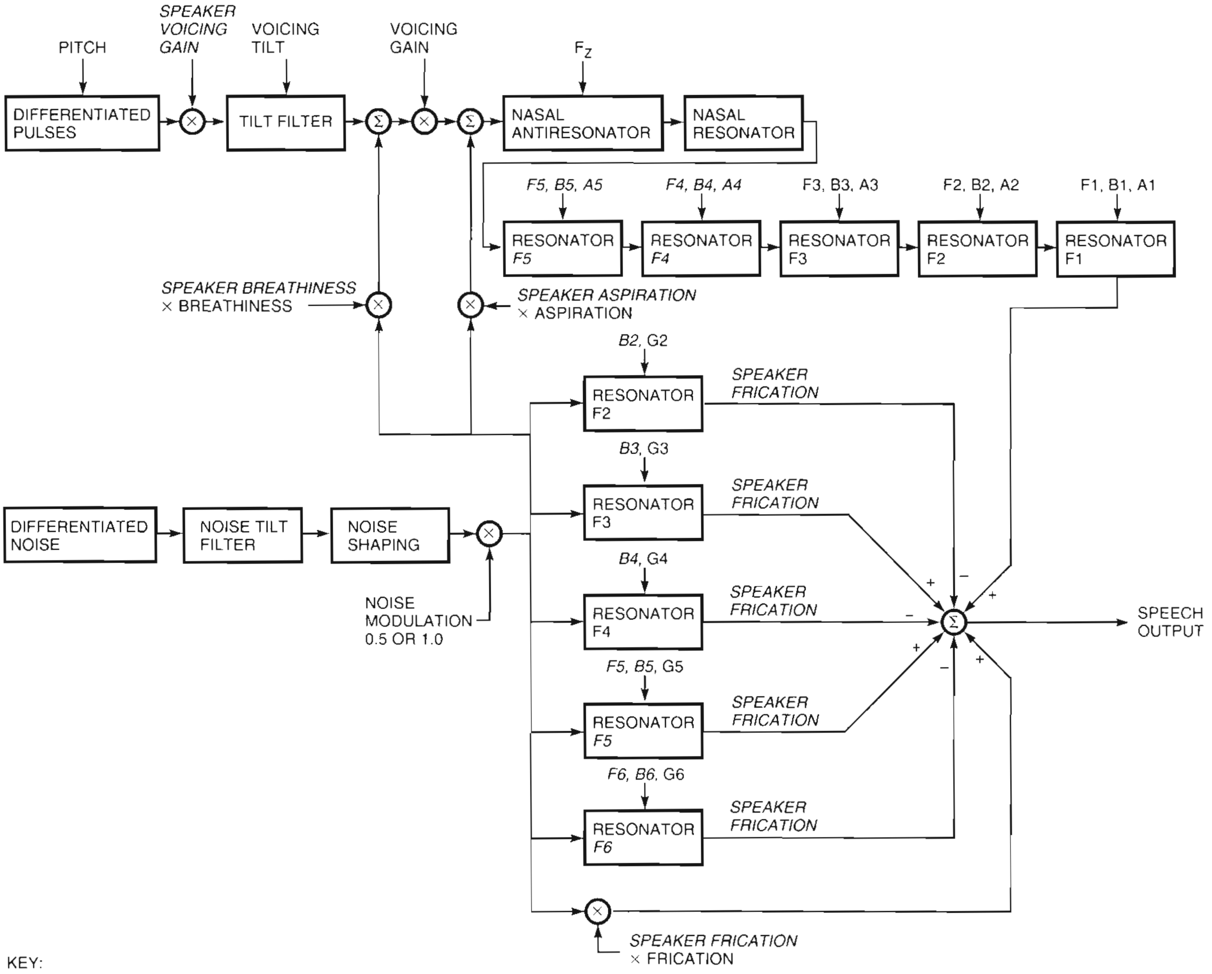
The radiation characteristic of the lips approximates a differentiation (derivative) of the acoustic pressure wave. Since all the filters in the VTM are linear and time-invariant, the radiation effects can be incorporated in the signal sources instead of at the output. Therefore the glottal source (pulse source) produces differentiated pulses. The differentiated noise signal is the filtered first difference of a uniform-distribution random number generator.

The DECtalk Software VTM (also known as the Klatt Synthesizer) is shown in Figure 7. The italicized terms are either speaker-dependent parameters or constant values. All other parameters are updated every frame. Depending on the system mode, the audio samples generated for each frame are passed to the output routine and subsequently are either queued to the audio device, written to a wave audio file, or written to a buffer provided by the application. After generating a speech frame, the VTM code increases the audio sample count by the frame size. This count is sent to the synchronization thread whenever a synchronization symbol or an index mark is received by the VTM thread. The count is reset to zero at startup and whenever the text-to-speech system is reset.

Tone messages are processed by the VTM thread. Tone messages are for single tones or DTMF signals. Each tone message includes two frequencies, two amplitudes (one for each frequency), and one duration. For a single tone message, the amplitude for the second frequency is zero. Tone synthesis code generates tone frames and queues them to the output routine. The first 2 milliseconds and the last 2 milliseconds of a tone signal are multiplied by either a rising or a falling cosine-squared shaping function to limit the out-of-band pulse energy. Each tone sample is synthesized using a sinusoid look-up table.[13]

The Synchronization Thread

The synchronization thread is idle unless the VTM thread forwards a synchronization symbol message or an index mark message. Both messages contain the current audio sample count. The index mark message



KEY:

F	FREQUENCY	A	AMPLITUDE
B	BANDWIDTH	G	GAIN

Note: Italicized terms are either speaker-dependent parameters or constant values.
All other parameters are updated every frame.

Figure 7
The DECtalk Software Vocal Tract Model (also known as the Klatt Synthesizer)

also contains an index mark number from 0 to 99. After receiving one of these messages, the synchronization thread periodically polls the audio thread until the indicated audio sample has been played. If the message contained a synchronization symbol, an event is set that unblocks the command thread. If it is an index mark message, the synchronization thread sends the index mark number back to the application. For the Digital UNIX operating system, this number is returned by calling a callback function that the application specifies when DECtalk Software is started. For the Windows NT operating system, the SendMessage function is used to return the index mark number to the application. The message is sent to a window procedure specified by the window handle that is provided when the text-to-speech system is started.

The Audio Thread

The audio thread manages all activities associated with playing audio through the computer's sound hardware. An audio API insulates DECtalk Software from the differences between operating systems. The audio API communicates with the audio thread. The VTM thread calls an audio API queuing function that writes samples to a ring buffer that is read only by the audio thread. The audio thread opens the audio device after approximately 0.8 seconds of audio samples have been queued and closes the audio device when there are no more samples to play. If the number of audio samples in the queue is too small to cause the audio device to be opened, and the flow rate (measured over a 100-millisecond interval) into the audio ring buffer is zero, the timer thread will send the audio thread a message that causes the audio device to open and start playing audio. When audio either starts or stops playing, a message is sent to the application.

For the Digital UNIX operating system, the audio thread is an interface to the low-level audio functions of the Multimedia Services for Digital UNIX (MMS) product. MMS provides a server to play audio and video.

For the Windows NT operating system, the implementation also uses the system low-level audio functions, but these functions interface directly with a system audio driver. The audio API provides capabilities to pause the audio, resume paused audio, stop audio from playing and cancel all queued audio, get the audio volume level, set the audio volume level, get the number of audio samples played, get the audio format, and set the audio format. An in-line play command can be used to play audio files. DECtalk Software uses the get format and set format audio capabilities to dynamically change the audio format so it can play an audio file that has a format different from the format generated by the VTM.

DECtalk Software API

In the mid-1980s, researchers at Digital's Cambridge Research Lab ported the DECtalk text-to-speech C language-based code to the ULTRIX operating system. The command, LTS, PH, and VTM portions of the system were different processes. The pipes were implemented using standard UNIX I/O handles, stdin and stdout. These, along with an audio driver process, were combined into a command procedure. This system lacked many of the rules and features found in DECtalk Software today, but it did demonstrate that real-time speech synthesis was possible on a workstation. Before this time, DECtalk required specialized Digital signal-processing hardware for real-time operation.[14] On a DECstation Model 5000/25 workstation, the text-to-speech implementation used 65 percent of the CPU. If the output sample rate of this system had been raised from 8,000 Hz to 11,025 Hz, the highest-quality rate provided by DECtalk Software, it would have loaded approximately 89 percent of the CPU. Workstation text-to-speech synthesis, while possible, was still very expensive.

The power of the Alpha CPU has changed this. Today, many copies of DECtalk Software can run simultaneously on Alpha-based systems. Speech synthesis is now a viable multimedia form. This change created the need for a text-to-speech API. Table 3 shows the DECtalk Software CPU load for various computers.

On Alpha systems, the performance of DECtalk Software depends primarily on the SPECmark rating of the computer. A lesser consideration is the secondary cache size. System bus bandwidth is not a limiting factor: The combined data rates for the text, phonemes, and audio are extremely low relative to modern bus speeds, even when running the maximum number of real-time text-to-speech processes that the processor can support.

The API we have developed is the result of collaboration between several organizations within Digital: the Light and Sound Group, the Assistive Technology Group, the Cambridge Research Lab, and the Voice

Table 3
DECtalk Software CPU Loading versus Processor SPECmarks

System	Clock (MHz)	Processor	Secondary Cache (MB)	SPECint92	SPECfp92	Audio Rate (kHz)	Total CPU Load (%)
Alpha AXP 150 PC	150	Alpha 21064	512	80.9	110.2	11,025	8
AlphaStation 250 4/266 workstation	266	Alpha 21064	2,048	198.6	262.5	11,025	2.4
DEC 3000 Model 800 workstation	200	Alpha 21064	2,048	138.4	188.6	11,025	5
DEC 3000 Model 900 workstation	275	Alpha 21064A	2,048	230.6	264.1	11,025	3
AlphaStation 400 4/233 workstation	233	Alpha 21064A	512	157.7	183.9	11,025	3
AlphaStation 600 5/266 workstation	266	Alpha 21164	2,048	288.6	428.6	8,000	1
XL 590 PC	90	Pentium	512	Unknown	N/A	11,025	24

and Telecom Engineering Group. We had two basic requirements: We wanted the API to be easy to use and to work with any text-to-speech system. While creating the API, we defined interfaces so that future improvements to the text-to-speech engine would not require any API calls to be changed. (Customers frown on product updates that require rewriting code.) Some decisions were controversial. Some contributors felt that the text-to-speech system should return speech samples only in memory buffers, and the application should shoulder the burden of interfacing to the workstation's audio subsystem. The other approach was to support the standard workstation audio (which is platform dependent) and to provide an API call that switched the system into a speech-to-memory mode. We selected the latter approach because it simplifies usage for most applications.

The API Functions

The core text-to-speech API functions are the TextToSpeechStartup function, the TextToSpeechSpeak function, and the TextToSpeechShutdown function. The simplest application might use only these three functions.

All applications using text-to-speech must call the TextToSpeechStartup function. This function creates all the DECtalk system threads and passes back a handle to the text-to-speech system. The handle is used in subsequent text-to-speech API calls. The startup function is the only API function that has different arguments for the Digital UNIX and the Windows NT operating systems. This is necessary because the asynchronous reporting mechanism is a callback function for Digital UNIX and is a system message for Windows NT. The TextToSpeechShutdown function frees all system resources and shuts down the threads. This would normally be called when closing the application.

The TextToSpeechSpeak function is used to queue text to the system. If an entire clause is not queued, no output will occur until the clause is completed by queuing additional text. A special TTS_FORCE parameter may be supplied in the function call to force a clause boundary. The TTS_FORCE parameter is necessary for applications that have no control over the text source and thus cannot guarantee that the final text forms a complete clause.

The text-to-speech API provides three audio output control functions. These pause the audio output (TextToSpeechPause), resume output after pausing (TextToSpeechResume), and reset the text-to-speech system (TextToSpeechReset). The reset function discards all queued text and stops all audio output.

The text-to-speech API also provides a special synchronization function (TextToSpeechSync) that blocks until all previously queued text has been spoken. This API call may not return for days if a sufficient amount of text is queued. (Index marks provide nonblocking synchronization.)

The API supplies functions to both load (TextToSpeechLoadUserDictionary) and unload (TextToSpeechUnloadUserDictionary) an application-defined dictionary. The dictionary contains words and their phonemic representations. The developer creates a dictionary using a window-based user-dictionary tool. This tool can speak words and their phonemic representations. It can also convert text sequences to phonemic sequences. This last feature frees the developer from having to memorize and use the DECtalk Software phonemic symbols.

Additional functions select the speaker voice, control the speaking rate, control the language, determine the system capabilities, and return status. The status API function can indicate if the system is currently speaking.

Special Text-to-Speech Modes

DECtalk Software has three special modes: the speech-to-wave file mode, the log-file mode, and the speech-to-memory mode. Each mode has two complementary calls, one to enter the mode and one to exit. When in the speech-to-wave file mode, the system writes all speech samples to a wave audio file. The file is closed when exiting this mode. This is useful on slower Intel systems that cannot perform real-time speech synthesis. The log-file mode causes the system to write the phonemic symbol output of the LTS thread to a file. The last mode is the speech-to-memory mode. After entering this mode, the application uses a special API call to supply the text-to-speech system with memory buffers. The text-to-speech system writes synthesized speech to these buffers and returns the buffer to the application. The buffers are returned using the same mechanism used for index marks, a callback function on the Digital UNIX operating system and a system message on the Windows NT operating system. These buffers may also return index marks and phonemic symbols and their durations. If the text-to-speech system is in speech-to-memory mode, calling the reset function causes all buffers to be returned to the application.

Porting DECtalk Software

The DECtalk PC code used a simple assembly language kernel to manage the threads. The existence of threads on our target platforms simplified porting the code. The thread functions, signals (such as conditions or events), and mutual exclusion objects are different for the Digital UNIX and the Windows NT operating systems. Since these functions occur mainly in the pipe code and the audio code, we maintain different versions of code for each system. The message-passing mechanism for Windows NT has no equivalent on Digital UNIX; therefore part of the API code had to be different. The command, LTS, and PH threads are all common code for Digital UNIX and Windows NT. Most of the VTM thread is also common code.

Porting the code for each thread required putting conditional statements that define thread entry points into each module for each supported operating system. We also had to add special code to each thread to support our API call that resets the text-to-speech system. The reset is the most complicated API operation, because the data piped between threads is in the form of variable-length packets. During a reset, it is incorrect to simply discard data within a pipe because the thread that reads the pipe will lose data synchronization. Therefore a reset causes each thread to loop and discard all input data until all the pipes are empty. Then each thread's control and state variables are set to a known state. In many complicated systems, resetting and shutting down are the most complicated parts of a control architecture. System designers should incorporate mechanisms to simplify these functions.

The VTM code is much shorter and simpler than the code in either the LTS or the PH thread, but it is by far the largest CPU load in the system. The DECtalk PC hardware used a specialized Digital Signal Processor (DSP) for the VTM. The research VTM code (written in the C language) was rewritten to be sample-rate-independent. The filters were all made in-line macros. With this new VTM, the DECtalk Software system loaded an Alpha AXP 150 PC product 31 percent. After rewriting this code using floating-point arithmetic and then converting it to assembly language, DECtalk Software loaded the processor less than 8 percent. (Both tests were conducted at an 11,025-Hz output sample rate.)

There are several reasons a floating-point VTM runs faster than an integer VTM on an Alpha system. An integer VTM requires a separate gain for each filter to keep the output data within the filter's dynamic range. For a floating-point VTM, the gains of all cascaded filters are combined into one gain. The increased dynamic range allows combining parts of some filters to reduce computations. Also, floating-point operations do not require additional instructions to perform scaling. The processor achieves greater instruction throughput because it can dual issue floating-point instructions with integer instructions, which are used for pointers, indices, and some loop counters. Finally, the current generation of Alpha processors performs some floating-point operations with less pipeline latency than their equivalent integer operations (note the SPECfp92 and SPECint92 ratings of the current Alpha processors listed in Table 3).

The integer VTM is faster than the floating-point VTM on Intel processors, so we maintain two versions of the VTM. Both versions support multiple sample rates. The pitch of the glottal source and the frequencies and bandwidths of the filters are adjusted for the output sample rate. When necessary, the filter gains are adjusted. These extra calculations do not add much to the total time used by the VTM because they are performed only once per frame.

Possible Future Improvements to DECtalk Software

The Assistive Technology Group continues to improve the letter-to-sound rules, the prosodic rules, and the phonetic rules. Future implementations could use object-oriented techniques to represent the dictionaries, words, phonemes, and parts of the VTM. A larger dictionary with more syntactic information can be added. There has even been some discussion of combining the LTS and PH threads to make more efficient use of lexical knowledge in PH. The glottal waveform generator can be improved. Syntactic parsers might provide the information required for more accurate intonation. Someday, semantic parsing (text understanding) may provide a major improvement in synthetic speech intonation. Researchers both within and outside of Digital are investigating these and many other areas. It seems likely that the American English version of DECtalk Software will continue to improve over time.

Summary

DECtalk Software provides natural-sounding, highly intelligible text-to-speech synthesis. It was developed to perform on the Digital UNIX operating system on Digital's Alpha-based platforms and with Microsoft's Windows NT operating system on both Alpha and Intel processors. It is based on the mature DECtalk PC hardware product. DECtalk Software also provides an easy-to-use API that allows applications to use the workstation's audio subsystem, to create wave audio files, and to write the speech samples to application-supplied memory buffers. An Alpha-based workstation can run many copies of DECtalk Software simultaneously.

DECtalk Software uses a dictionary and linguistic rules to convert speech to phonemes. An application-supplied dictionary can override the default pronunciation of a word. Prosodic and phonetic rules modify the phoneme's attributes. A vocal tract model synthesizes each phoneme to produce a speech waveform. The result is the highest-quality text to speech. The Assistive Technology Group continues to improve the DECtalk text-to-speech algorithms.

Acknowledgments

I wish to acknowledge and thank all the members of the DECtalk Software project and additional support staff. Bernie Rozmovits, our engineering project leader, was the visionary for this entire effort. He contributed most of our sample applications on Windows NT, and he also wrote the text-to-speech DDE server. Krishna Mangipudi, Darrell Stam, and Hugh Enxing implemented DECtalk Software on the Digital UNIX operating system. Thanks to Bill Scarborough who did a great job on all of our documentation, particularly the on-line help. Special thanks to Dr. Tony Vitale and Ed Bruckert, from Digital's Assistive Technology Group. They both were instrumental in developing the DECtalk family of products and are continuing to improve them. Without their efforts and support, DECtalk Software could not exist. Tom Levergood and T. V. Raman at Digital's Cambridge Research Lab helped test DECtalk Software and provided many suggestions and improvements. Thanks also to the engineering manager for Graphics and Multimedia, Steve Seufert, who continues to support our efforts. Finally, we are all indebted to Dennis Klatt who was the creator of the DECtalk speech synthesizer and to all the other developers of the original DECtalk hardware products.

References

1. G. Fant, *Acoustic Theory of Speech Production* (The Netherlands: Mouton and Co. N.V., 1960).

2. C. Schmandt, *Voice Communication with Computers* (New York: Van Nostrand Reinhold, 1994).

3. J. Allen, M. Hunnicutt, and D. Klatt, *From Text to Speech: The MITalk System* (Cambridge, Mass.: Cambridge University Press, 1987).

4. J. Flanagan, *Speech Analysis, Synthesis, and Perception*, 2d ed. (New York: Springer-Verlag, 1972).

5. D. Pisoni, H. Nusbaum, and B. Greene, "Perception of Synthetic Speech Generated by Rule," *Proceedings of the IEEE*, vol. 73, no. 11 (1985): 1665–1676.

6. A. Vitale and M. Divay, "Algorithms for Grapheme-Phoneme Translation in French and English" (in preparation).

7. V. Fromkin and R. Rodman, *An Introduction to Language*, 2d ed. (New York: Holt, Rinehart, and Winston, 1978).

8. D. Klatt, "Review of Text-to-Speech Conversion for English," *Journal of the Acoustical Society of America*, vol. 82, no. 3 (1987): 737–793.

9. D. Klatt, "Software for a Cascade/Parallel Formant Synthesizer," *Journal of the Acoustical Society of America*, vol. 67 (1980): 971–975.

10. L. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing* (London: Prentice Hall, 1975).

11. L. Rabiner and R. Schafer, *Digital Processing of Speech Signals* (London: Prentice Hall, 1978).

12. D. Klatt and L. Klatt, "Analysis, Synthesis, and Perception of Voice Quality Variations among Female and Male Talkers," *Journal of the Acoustical Society of America*, vol. 87, no. 2 (1990): 820–857.

13. J. Tierney, "Digital Frequency Synthesizers," Chapter V of *Frequency Synthesis: Techniques and Applications*, J. Gorski-Popel, ed. (New York: IEEE Press, 1975).

14. E. Bruckert, M. Minow, and W. Tetschner, "Three-Tiered Software and VLSI Aid Development System to Read Text Aloud," *Electronic* (April 21, 1983).

Biography

William I. Hallahan
Bill Hallahan is a member of the Light and Sound Group, part of Software Engineering for the Workstation Business Segment. Previously he worked in the Image, Voice, and Video Group on signal-processing algorithms and the rewriting of the DECtalk vocal tract model. Before joining Digital in 1992, he was employed at Sanders Associates for 12 years, where he developed and implemented algorithms that performed signal analysis, signal demodulation, and numerical methods. Bill received a B.S.E.E. from the University of New Hampshire in 1980. He is co-author of a patent application for a specific color-space conversion algorithm used in video multimedia applications.

Kenneth W. Correll
Robert A. Ulichney

The J300 Family of Video and Audio Adapters: Architecture and Hardware Design

The J300 family of video and audio adapters provides a feature-rich set of hardware options for Alpha-based workstations. Unlike earlier attempts to integrate full-motion digital video with general-purpose computer systems, the architecture and design of J300 adapters exploit fast system and I/O buses to allow video data to be treated like any other data type used by the system, independent of the graphics subsystem. This paper describes the architecture used in J300 products, the video and audio features supported, and some key aspects of the hardware design. In particular, the paper describes a simple yet versatile color-map-friendly rendering system that generates high-quality 8-bit image data.

The overall architectural design goal for the J300 family of video and audio adapters was to provide the hardware support necessary to allow the integration of broadcast video into workstations. The three primary objectives were as follows: (1) digitized video data should be treated the same as any other data type in the system; (2) the video and the graphics subsystem designs should be completely independent of each other; and (3) any hardware designed should be low cost.

Digital has implemented the J300 architecture in three products: Sound & Motion J300, FullVideo Supreme JPEG, and FullVideo Supreme.[1] The Sound & Motion J300 (referred to in this paper simply as the J300) was the first product designed with this architecture and is the primary focus of this paper. The FullVideo Supreme JPEG and FullVideo Supreme products are based on the same design database as the J300. They differ from the J300 in the bus supported (they support the peripheral component interconnect [PCI] bus) and the lack of audio support. Additionally, the FullVideo Supreme product does not include hardware compression/decompression circuitry.

The J300 brings a wide range of video and audio capabilities to machines based on Digital's TURBOchannel I/O interconnect. Analog broadcast video can be digitized, demodulated, and rendered for display on any graphics device. The J300 provides hardware video compression and decompression to accelerate applications such as videoconferencing. The J300 supports analog broadcast video output from either compressed or uncompressed video files. Audio support includes a general-purpose, digital signal processor (DSP) to assist in the real-time management of the audio streams and for advanced processing, such as compression, decompression, and echo cancellation. Audio input and output capabilities include stereo analog I/O, digital audio I/O, and a headphone/microphone jack. Analog audio can be digitized to 16 bits per sample at a rate of up to 48 kilohertz (kHz).

This paper begins with an overview of some terminology commonly used in the field of broadcast video. The paper then presents the evolution and design of the J300 architecture, including several key enabling technologies and the logical video data paths available. Next follows a discussion of the hardware design phase of the project and the trade-offs made to reconcile expectation and implementation. Detailed descriptions are devoted to specific areas of the design, including the video I/O logic, the AccuVideo rendering path, and the video and audio direct memory access (DMA) interfaces.

Video Terminology Overview

Three fundamental standards are in use worldwide for representing what is referred to in this paper as broadcast video: the National (U.S.) Television System Committee (NTSC) recommendation, Phase Alternate Line (PAL), and Séquentiel Couleur avec Mémoire (SECAM). The standards differ in the number of horizontal lines in the display, the vertical refresh rate, and the method used for encoding color information. North America and Japan use the 525-line, 60-hertz (Hz) NTSC format; PAL is used in most of Europe; and SECAM is used primarily in France. Both the PAL and SECAM standards are 625-line, 50-Hz systems.[2]

All three television standards split an image or a frame of video data into two fields, referred to as the even and the odd fields. Each field contains alternate horizontal lines of the frame. The vertical refresh rate cited in the previous paragraph is the field rate; the frame rate is one-half of that rate.

Unlike computer display systems that use red, green, and blue (RGB) signals to represent color information, PAL and SECAM use a luminance-chrominance system, which has the three parameters Y (the luminance component), and U and V (the two chrominance components). NTSC uses a variation of YUV, where the U and V components are rotated by 33 degrees and called I and Q. YUV is related to RGB by the following conversion matrix:[3]

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

All the different standards limit the bandwidth of the chrominance signal to between one-quarter and one-third that of the luminance signal. This limit is taken into account in the digital representation of the signal and results in what is called 4:2:2 YUV, where, for every four horizontally adjacent samples of Y, there are two samples of both U and V. All three components are sampled above the Nyquist rate in this format with a significant reduction in the amount of data needed to reconstruct the video image.

Various modulation techniques transform the separate Y, U, and V components into a single signal, typically referred to as composite video. To increase the fidelity of video signals by reducing the luminance-chrominance cross talk caused by modulation, the S-Video standard has been developed as an alternative. S-Video, which refers to separate video, specifies that the luminance signal and the modulated chrominance signal be carried on separate wires.

The J300 includes hardware support for the Joint Photographic Experts Group (JPEG) compression/decompression standard.[4] JPEG is based on the discrete cosine transform (DCT) compression method for still-frame color images. DCT is a widely accepted method for image compression because it provides an efficient mechanism to eliminate components of the image that are not easily perceived by casual inspection.

Design History and Motivation

Digital arrived at the J300 adapter design after considering several digital video playback architectures. The Jvideo advanced development project, the implementation of one of the alternatives, was instrumental in achieving the design goals.

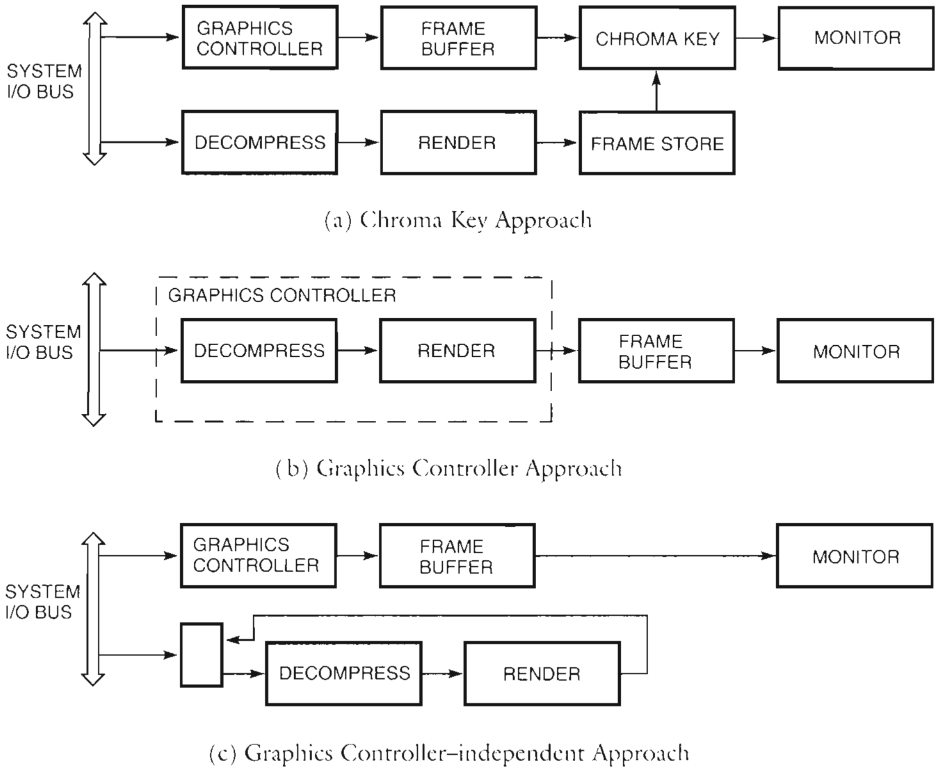
Architectural Alternatives and Objectives

In January 1991, several Digital engineering organizations collaborated to define the architecture of a hardware seed project that could be used to explore a workstation's capability to process video data. The participants felt that the key technologies required to explore the goal of integrating computers and broadcast video were available. These enabling technologies were

1. The TURBOchannel high-speed I/O bus, which was a standard on Digital workstations
2. The anticipated acceptance of the JPEG compression/decompression standard and single-chip implementations that supported that standard
3. The development of a rendering system (now called the AccuVideo system) that could map YUV input values into an 8-bit color index using any number of available colors with very good results

We evaluated the three alternative approaches shown in Figure 1 for moving compressed video data from system memory, for decompressing and rendering the data, and, finally, for moving the data into the frame buffer.

The chroma key approach, shown in Figure 1a, differs little from previous work done at Digital and was the primary architecture used by the industry. Several variations of the exact implementation are in use, but, basically, the graphics device paints a designated color into sections of the frame buffer where the video data is to appear on the display. A comparator located between the graphics frame buffer and the display device looks at the serial stream of data coming from the graphics frame buffer and, when the data matches the chroma key (stored in a register), inserts the video data. As shown in Figure 1a, this approach



(a) Chroma Key Approach

(b) Graphics Controller Approach

(c) Graphics Controller–independent Approach

Figure 1
Digital Video Playback Architectures

relies on a special connection between the video decompression block and the output of the graphics device. While this approach off-loads the system I/O bus, it treats video data differently from other data types to be displayed. In particular, the X Window System graphical windowing environment has no knowledge of the actual contents of the video window at any given time.

The graphics controller approach, shown in Figure 1b, integrates the decompression technology with the graphics accelerator. Although this approach has the potential of incurring the lowest overall system cost, it fails in two important aspects. First, it does not expose the windowing system to the video data. Second, since the graphics controller and video logic are integrated, the user must accept the level of graphics performance provided. No graphics upgrade path exists, so upgrading would require another product development cycle. Including the video logic across the range of graphics devices is not desirable, because such a design forces higher prices for users who are not interested in the manipulation of broadcast video.

The third approach, shown in Figure 1c, is much more radical. It places the responsibility of moving each field of video data to and from the decompression/rendering option squarely on the system. The system I/O bus must absorb not only the traffic generated by the movement of the compressed video to the decompression hardware but also the movement of the decompressed video image from the accelerator back to system memory and back again over the same bus to the graphics option.

Accepting the third alternative architecture allowed us to meet the three important objectives for the project:

1. The workstation should be able to treat digitized video data the same as any other data type.
2. The inclusion of video capabilities in a workstation should be completely independent of the graphics subsystem used.
3. Any hardware option should be low cost.

The original design goals included audio I/O, even though the processing power and bandwidth needed for audio were far below those required for video. Since users who want video capability usually require audio capability as well, audio support was included so that users would have to buy only one option to get both audio and video. This design reduced the number of bus slots used.

The Jvideo Advanced Development Project

Jvideo was the name given to the advanced development hardware seed project. Actual design work started in February 1991; power on occurred in September 1991. Jvideo has since become a widely used research tool.

Table 1
The Nine Video Flow Paths

	Output			
Input	Analog	Compressed	Uncompressed	Dithered
Analog	...	A → C	A → U	A → D
Compressed	C → A	...	C → U	C → D
Uncompressed	U → A	U → C	...	U → D

Jvideo was an important advanced development project for several reasons. First, it was the vehicle used to verify the first two project objectives. Second, it was the first complete hardware implementation of the rendering circuit, thus verifying the image quality that was available when displaying video with fewer than 256 colors. Finally, it was during the development of Jvideo that the DMA structure and interaction with the system was developed and verified.

J300 Features

This section describes the various video paths supported in the J300 and presents videoconferencing as an example of video data flow. The AccuVideo filter-and-scale and dithering system designs used in the J300 are presented in detail.

Video Paths

Table 1 summarizes the nine fundamental video paths that the J300 system supports. The input to the J300 can come from an external analog source or from the system in compressed or uncompressed form. The outputs include analog video and several internal formats, i.e., JPEG compressed, uncompressed, or dithered. Dithering is a technique used to produce a visually pleasant image while using far less information than was available in the original format.

A conceptual flow diagram of the major components of the J300 video system is shown in Figure 2. Physically, the frame store and the blocks to its left make up the video board. All the other blocks except for JPEG compression/decompression are part of the J300 application-specific integrated circuit (ASIC). (The J300 Hardware Implementation section provides details on this ASIC.)

Both the upscale prior to the analog out block and the downscale after the analog in block scale the image size independently in the horizontal and vertical directions with arbitrary real-value scale factors. The filter-and-downscale function is handled by the Philips chip set, as described in the J300 Hardware Implementation section. The upscale block is a copy of the Bresenham-style scale circuit used in the filter-and-scale block.

The Bresenham-style scale circuit is extremely simple and is described in "Bresenham-style Scaling," along with an interesting closed-form solution for finding initial parameters.[5] The filter-and-scale block is part of the J300 rendering system. The J300 supports arbitrary scaling for either enlargement or reduction in both dimensions. We carefully selected a few simple, three-element horizontal filters to be used in combination with scaling; the filters were small enough to be included in the J300 ASIC. The J300 supports three sharpening filters that are based on a digital Laplacian:[6]

Low sharpness	(−1/2	2	−1/2)
Medium sharpness	(−1	3	−1)
High sharpness	(−2	5	−2)

The J300 also supports two low-pass or smoothing filters:

Low smoothing	(1/4	1/2	1/4)
High smoothing	(1/2	0	1/2)

Sharpening is performed before scaling for enlargement and after scaling for reduction. Smoothing is always performed before scaling (as a band limiter) for reduction and after scaling (as an interpolator) for enlargement.

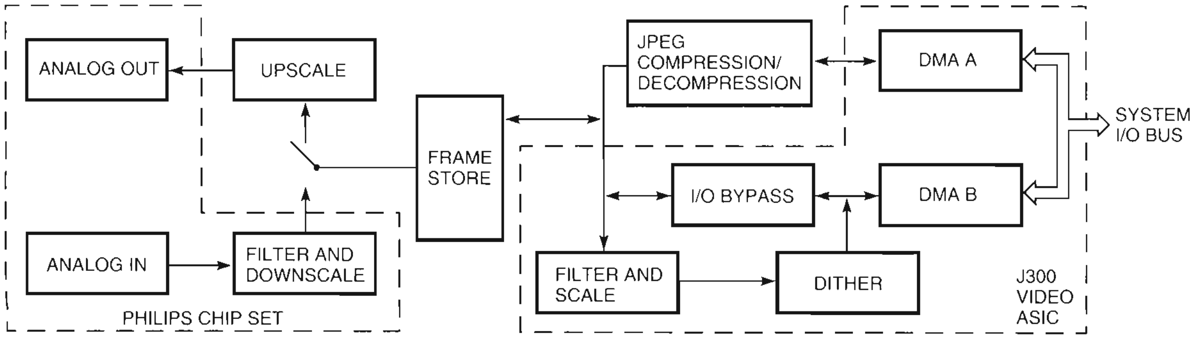


Figure 2
J300 Video Flow

The second part of video rendering occurs in the dither block. The AccuVideo Rendering section provides details on this block.

The I/O bypass skips over the video rendering blocks when undithered uncompressed output is required. When uncompressed digital video in used as input, the I/O bypass is also used. DMA B thus passes dithered or uncompressed output and uncompressed input.

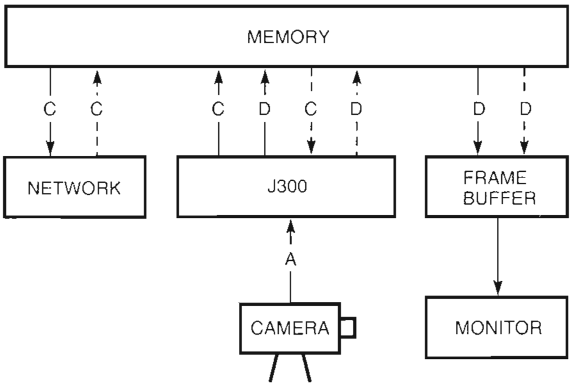
Compressed input and compressed output are passed through DMA A. The JPEG compression/decompression block handles all compression of output and decompression of input. The combination of the two DMA channels allows high data rates because both channels are often used in parallel.

Videoconferencing Application

A good illustration of the video data flow in J300 is a videoconferencing application. Figure 3 shows the flow of analog (A), compressed (C), and dithered (D) video data to and from memory in a system on a network. The application software controls the flow of data between memory and the display and network devices. The J300 hardware must perform two fundamental operations:

1. Capture the local analog signal, compress the data, and send it to memory, and in parallel, dither the data and send it to memory. The solid arrows in Figure 3 denote the compress, send, and view paths.
2. Receive a remote compressed video stream from memory, decompress and dither the data, and send it back to memory. The dashed arrows in Figure 3 denote the receive, decompress, and view paths.

Figure 3 demonstrates the unique graphics controller independence of the J300 architecture, as shown in Figure 1c. In assessing the aggregate video data traffic, it is important to keep in mind that the dithered data is 8 bits per pixel, and the compressed data is approximately 1.5 bits per pixel. For example, consider a videoconference with 11 participants, where each person's workstation screen displays the images of the other 10 participants, each in a 320-by-240-pixel window and with a refresh rate of 20 Hz. The bus traffic required for each window is twice the compressed image size plus twice the decompressed image size, i.e., (2 × 320 × 240 × 1.5) ÷ 8 bytes + (2 × 320 × 240) bytes = 182.4 kilobytes (kB) per window. The total bandwidth would be 182.4 kB × 11 windows × 20 Hz = 40.1 megabytes (MB) per second, which is well within the achievable bandwidth of both TURBOchannel and PCI buses.



Notes: Dashed arrows represent the receive, decompress, and view paths (C→D). Solid arrows represent the compress, send, and view paths (A→C, A→D). The symbols A, C, and D stand for analog, compressed, and dithered data.

Figure 3
Videoconferencing Application

These two operations through the J300 conceptual flow diagram of Figure 2 are shown explicitly in Figure 4 for the capture, compress, and dither paths, and in Figure 5 for the decompress and dither path. In Figure 4, video data is captured through the analog in block and buffered in the frame store block. The frame store then sends the data in parallel to the JPEG compression/decompression path, and to the filter, scale, and dither path, each of which sends the data to its own dedicated DMA port.

In Figure 5, compressed data enters DMA A, is JPEG decompressed using the frame store as a buffer, and is sent to the filter, scale, and dither path, where it is output through DMA B.

Figures 4 and 5 illustrate three of the nine possible video paths shown in Table 1. It is straightforward to see how the other six paths flow through the block diagram of Figure 2.

AccuVideo Rendering

Digital's AccuVideo method of video rendering is used in the J300 and in other products.[7,8] J300 rendering is represented in Figure 2 by the filter-and-scale block and by the dither block. The following features are supported:

- High-quality dithering
- Selectable number of colors from 2 to 256
- YUV-to-RGB conversion with controlled out-of-bounds mapping
- Brightness, contrast, and saturation control
- Color or gray-scale output
- Two-dimensional (2-D) scaling to any size
- Sharpening and smoothing control

The algorithm for mean-preserving multilevel dithering is described by Ulichney in "Video Rendering."[9] Mean preserving denotes that the macroscopic average in the output image is maintained across the entire range of input values. Figure 6 depicts the version of the dithering algorithm used for the single component Y in the J300 prototype, Jvideo.

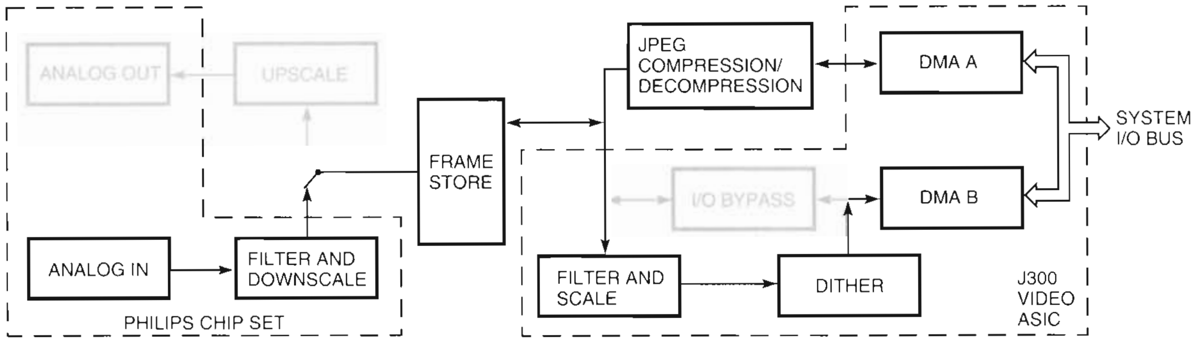


Figure 4
Capture, Compress, and Dither Paths

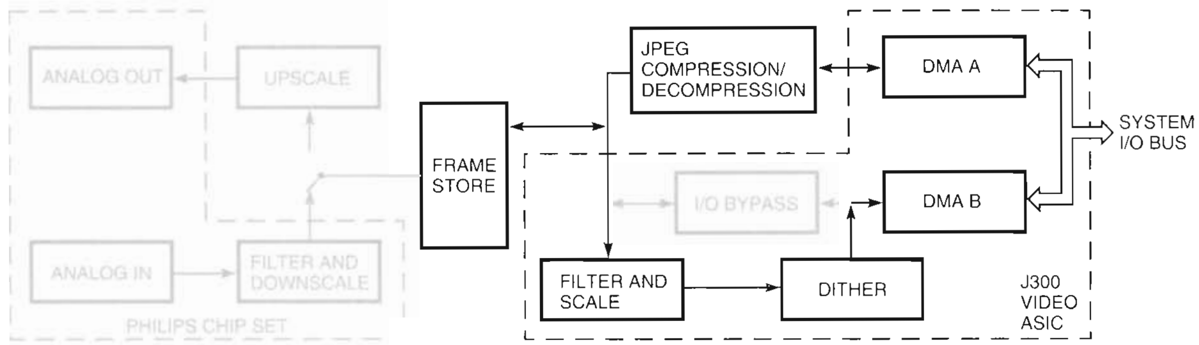


Figure 5
Decompress and Dither Path

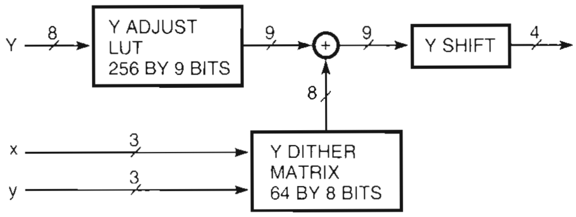


Figure 6
Dither Components of the Jvideo Prototype

To quantize with a simple shift register and still maintain mean preservation, a particular gain that happens to have a value between 1 and 2 must be imparted to the input.[9] This gain is included in the adjust look-up table (LUT), thus adding a bit to the data width of the input value to the ditherer.

In the case of the *Y* (luminance) component, the effect of brightness and contrast can be controlled by dynamically changing and loading the contents of this adjust LUT. Saturation control is a contrast-like mapping controlled on the *U* and *V* adjust LUTs.

The least significant bits of the horizontal and vertical address (*x*,*y*) of the pixel index the dither matrix. In the Jvideo prototype, we used an 8 by 8 recursive tessellation array.[7] Because the size of the array was so small, all the components in Figure 6 could be encapsulated with a single 16K-by-4-bit random-access memory (RAM). This implementation is not the least expensive, but it is the easiest to build and is quite appropriate for a prototype.

Figure 7 illustrates the Jvideo dither system. The number of dither levels and associated color adjustment are designed in software and loaded into each of the 16K-by-4-bit LUTs for *Y*, *U*, and *V*. Each component outputs from 2 to 15 dithered levels. The three 4-bit dithered values are used as a collective address to a color convert LUT, which is a 4K-by-8-bit RAM.

Loaded into this LUT is the conversion of each YUV triplet to one of *N* RGB index values. The generation of this LUT incorporates the state of the display server's color map at render time. Although this approach is much more efficient than a direct algebraic conversion known as dematrixing, an arbitrarily complex mapping of out-of-range values can take place because the table is built off line. Another paper in this issue of the *Journal*, "Software-only Compression, Rendering, and Playback of Digital Video," presents details on this approach.[7]

Perhaps the central characteristic of AccuVideo rendering is the pleasing nature of the dither patterns generated. We are able to obtain such patterns because we incorporate dither matrices developed using the void-and-cluster method.[10] These matrices are 32 by

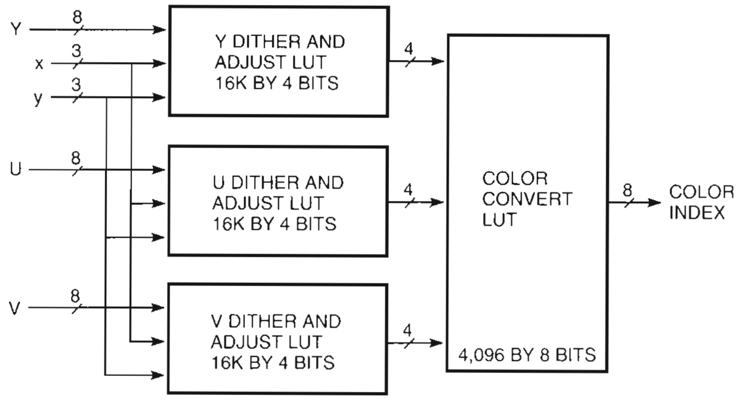

Y
8
x
3
y
3
Y DITHER AND ADJUST LUT 16K BY 4 BITS
4
U
8
U DITHER AND ADJUST LUT 16K BY 4 BITS
4
V
8
V DITHER AND ADJUST LUT 16K BY 4 BITS
4
COLOR CONVERT LUT
4,096 BY 8 BITS
8
COLOR INDEX

Figure 7
Jvideo Dither System

32 in extent. Although surprisingly small for the complexity and seamlessness of the patterns produced, this size requires 10 bits of display address information for indexing.

While very simple to implement, the single LUT approach used in the Jvideo system shown in Figure 7 becomes unattractive for a matrix of this size because of the large memory requirement. Eight bits of input plus 10 bits of array address requires a 256K-bit RAM for each color component; Jvideo's 8 by 8 dither matrix called for a more cost-effective 16K-bit RAM.

The dither system design used in the J300 is shown in Figure 8. The design is quite simple, requiring only RAM and three adders. We restricted the number of *U*- and *V*-dithered levels to always be equal. Such a restriction allows the sharing of a single dither matrix RAM. The paper "Video Rendering" provides details on the relationship between the number of dithered levels for each component, the number of bits shifted, the normalization of the dither matrix values, the gain embedded in the adjust LUT, and the bit widths of the data paths.[9] Note that the decision to use RAM instead of read-only memory (ROM) for the adjust LUTs, dither matrices, and color convert LUT permits complete flexibility in selecting the number of dithered colors.

When the video source is monochrome, or whenever a monochrome display is desired, a Mono Select mode allows the *Y* channel to be quantized to up to 8 bits.

The algorithm used in the software-only version of AccuVideo exactly parallels Figure 8.[7] "Integrating Video Rendering into Graphics Accelerator Chips" describes variations of this architecture for other products.[8] One design always renders the same number of colors without adjustment, in favor of very low

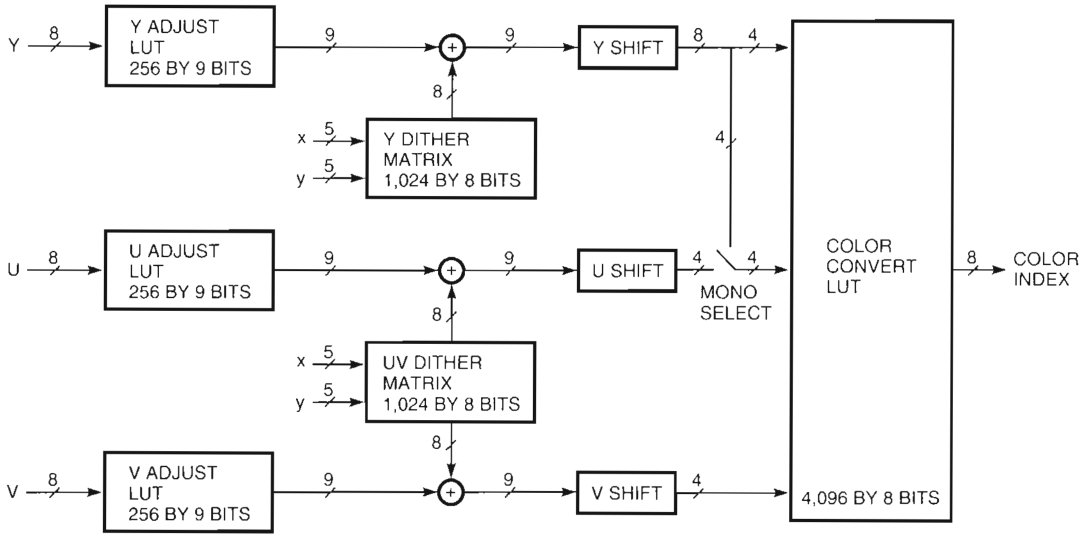

Y
8
Y ADJUST LUT 256 BY 9 BITS
9
9
Y SHIFT
8
4
8
x
5
y
5
Y DITHER MATRIX 1,024 BY 8 BITS
4
U
8
U ADJUST LUT 256 BY 9 BITS
9
9
U SHIFT
4
4
MONO SELECT
8
x
5
y
5
UV DITHER MATRIX 1,024 BY 8 BITS
8
V
8
V ADJUST LUT 256 BY 9 BITS
9
9
V SHIFT
4
COLOR CONVERT LUT
4,096 BY 8 BITS
8
COLOR INDEX

Figure 8
J300 Dither System

cost. Another performs YUV-to-RGB conversion first, to allow dithering to more than 256 colors. Note that with this design, for large numbers of output colors, the memory required for the back-end color convert LUT design would be prohibitive.

J300 Hardware Implementation

Implementing the J300 hardware design entailed making trade-offs to keep down the costs. This section presents the major trade-offs and then discusses the resulting video and audio subsystem designs, the built-in I/O test capabilities, and the Verilog hardware description language design environment used.

Design Trade-offs

In August 1991, the Jvideo hardware design team presented to engineering management several cost-reducing design alternatives with the goal of turning Jvideo into a product. Alternatives ranged from retaining the basic design (which would require a short design time and would result in the fastest time to market) to redesigning the board with minimal cost as the driving factor (which meant putting as much logic as possible into the J300 ASIC). Management accepted the latter proposal, and design started in January 1992.

The major design trade-offs involved in reducing module cost centered around three portions of the design: the accelerator chip, the pixel representation, and the dither circuit. The design team evaluated different JPEG hardware compression/decompression accelerators in terms of availability, performance, cost, and schedule risk. While various manufacturers claimed to have cheaper parts available within our design schedule constraints, the CL550 chip from C-Cube Microsystems, the same chip used in the Jvideo system, had reasonable performance and known idiosyncrasies. The designers decided to use one CL550 chip instead of two, as was done in Jvideo. This meant that in videoconferencing applications, the chip would have to be programmed to compress the incoming image and then reprogrammed to decompress the other images. The turnaround time of the programming required to implement the design change plus the compression time together accounted for the performance penalty that the product would pay for including only one CL550.

To understand the impact on performance of using just one CL550 chip, consider that all 700 registers in the chip would have to be reloaded when changing the chip from compression to decompression and vice versa. Given a register write cycle of 250 nanoseconds, the penalty is 175 microseconds. We estimated the time to compress an image as the number of pixels in the uncompressed image (the CL550 does occasionally stall during compression or decompression, but we ignored this fact for these calculations) times the period of the pixel rate. For an image size of 320 by 240 pixels and a pixel clock period of 66.67 nanoseconds, the time used for compression is 5.12 milliseconds. If the desired overall frame rate of all images on the screen is 20 Hz, then approximately 11 percent of the available time is given to compression ((5.12 milliseconds + 0.35 milliseconds) ÷ 50 milliseconds). We judged this decrease in decompression performance reasonable, since approximately 30 percent of the early estimated cost of materials on the J300 was the CL550 and the associated circuits.

The second major area of savings came with the decision to use the 4:2:2 YUV pixel representation in the frame store, the CL550, and the input to the rendering logic. This approach reduced the width of the frame store and external data paths from 24 to 16 bits with no loss of fidelity in the image. The trade-off associated with this decision was that the design precluded the ability to directly capture video in 24-bit RGB unless the ASIC included a full YUV-to-RGB conversion. The main thrust of the product was to accelerate image compression and decompression on what was assumed to be the largest market, i.e., 8-bit graphics systems, by using the AccuVideo rendering path. Since 24-bit RGB can be obtained from 4:2:2 YUV pixel representation (which can be captured directly) with no loss of image fidelity, we considered this hardware limitation to be minor.

The third area of trade-offs revolved around the implementation of the dither circuit and how much of that circuitry the ASIC should include. The rendering system on Jvideo was implemented entirely with LUTs, a method that is inexpensive in terms of the random logic needed but expensive in terms of component cost. Early on, the design team decided that including the 4K-by-8-bit color convert LUT inside the ASIC was not practical. Placing the LUT outside the ASIC required using a minimal number of pins, 28, and using a readily available 8K-by-8-bit static random-access memory (SRAM) allowed the unused portion of the RAM to store the dither matrix values. Such a design reduced the amount of on-chip storage required for dither matrix values to 32 by 8 bits.

The impact of requiring dither matrix value fetches on a per-line basis added to the interline overhead 32 accesses for the new dither matrix values or 16 pixel clocks. The impact of the 16 added clocks on a line basis depends on the resultant displayed image size. If the displayed images are small, the impact is as much as 10 percent (for a 160-by-120-pixel image). It is uncommon, however, for someone to view video on a workstation at that resolution. At a more common displayed size of 640 by 480, the amount of overhead decreases to 3 percent.

Video Subsystem Design

The major elements of the video subsystem design are the ASIC, which is designed in the Verilog hardware description language, the Philips digital video chip set, and the compression/decompression circuitry. This section discusses the ASIC design and some aspects of the video I/O circuit design.

The J300 ASIC The J300 ASIC design included not only the video paths discussed earlier in the section J300 Features but also all the control for the video I/O section of the design, all video random-access memory (VRAM) control, the CL550 interface, access to the diagnostics ROM, arbitration with the audio circuit for TURBOchannel access, and the TURBOchannel interface. Figure 9 shows a block diagram of the J300 ASIC. Only the DMA section of the design is discussed further in this paper.

The DMA interface built into the ASIC is designed to facilitate the movement of large blocks of data to or from system memory with minimal interaction from the system. The chip supports two channels: the first is used for CL550 host port data (compressed video and register write data); the second is used for pixel data flowing to or from the rendering circuit. Once started, each channel uses its map pointer register to access successive (*address, length*) pairs that describe the physical memory to be used in the operation. (The map pointer register points to the scatter/gather map in system memory to be used.) The ASIC fills or empties the first buffer and then automatically fetches the next (*address. length*) pair in the scatter/gather map and so on until the operation is complete. When a compressed image is transferred into system memory, the exact length of the data set is unknown until the ASIC detects the end-of-image marker from the CL550. In this case, system software can read a length register to find out exactly how much data was transferred.

There is no restriction on the number of (*address. length*) pairs included in each scatter/gather map. New pairs can be assigned to each line of incoming video such that deinterlacing even and odd video fields can be accomplished as the data is moved into system memory.

Since only the map pointer register needs to be updated between operations, system software can set up multiple buffers, each with its associated scatter/gather map, ahead of time.

Video Input and Output Logic The J300 video I/O circuit, shown in Figure 10, was designed using Philips Semiconductors' digital video chip set. Explanation of some aspects of the design follows.

The J300 uses the Philips chip set to digitize and decode input video. The chip set consists of the TDA8708A and the TDA8709A, as the analog-to-digital (A/D) converters, and the SAA7191, as the Digital MultiStandard Decoder (DMSD). This chip

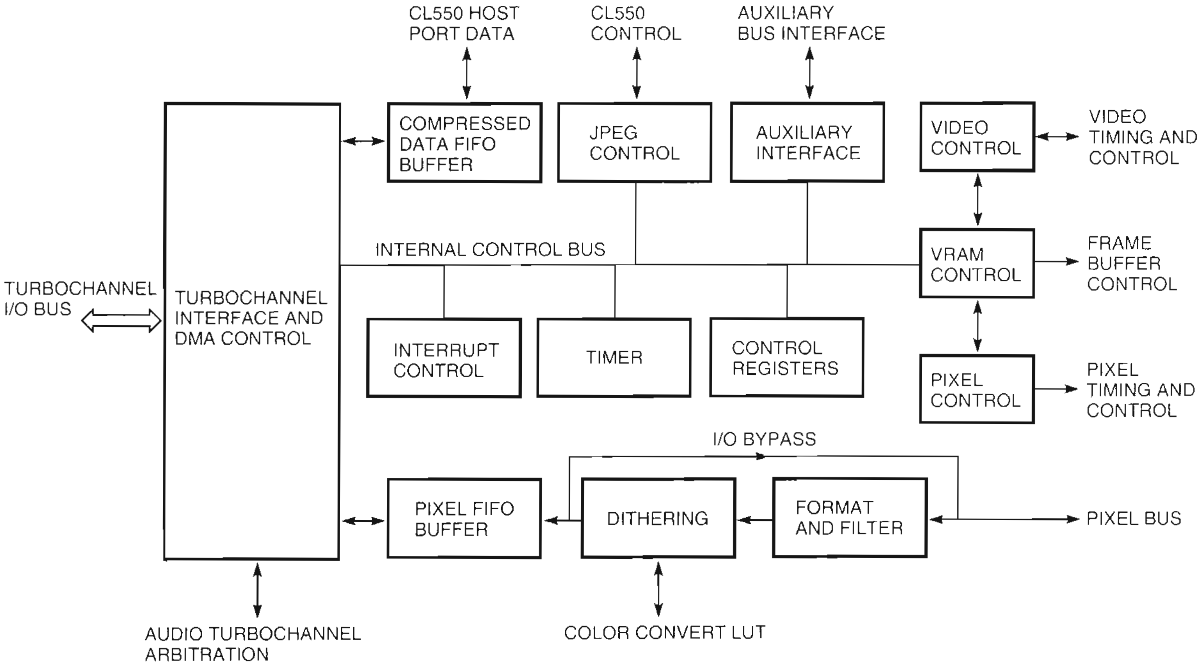


Figure 9
J300 ASIC Block Diagram

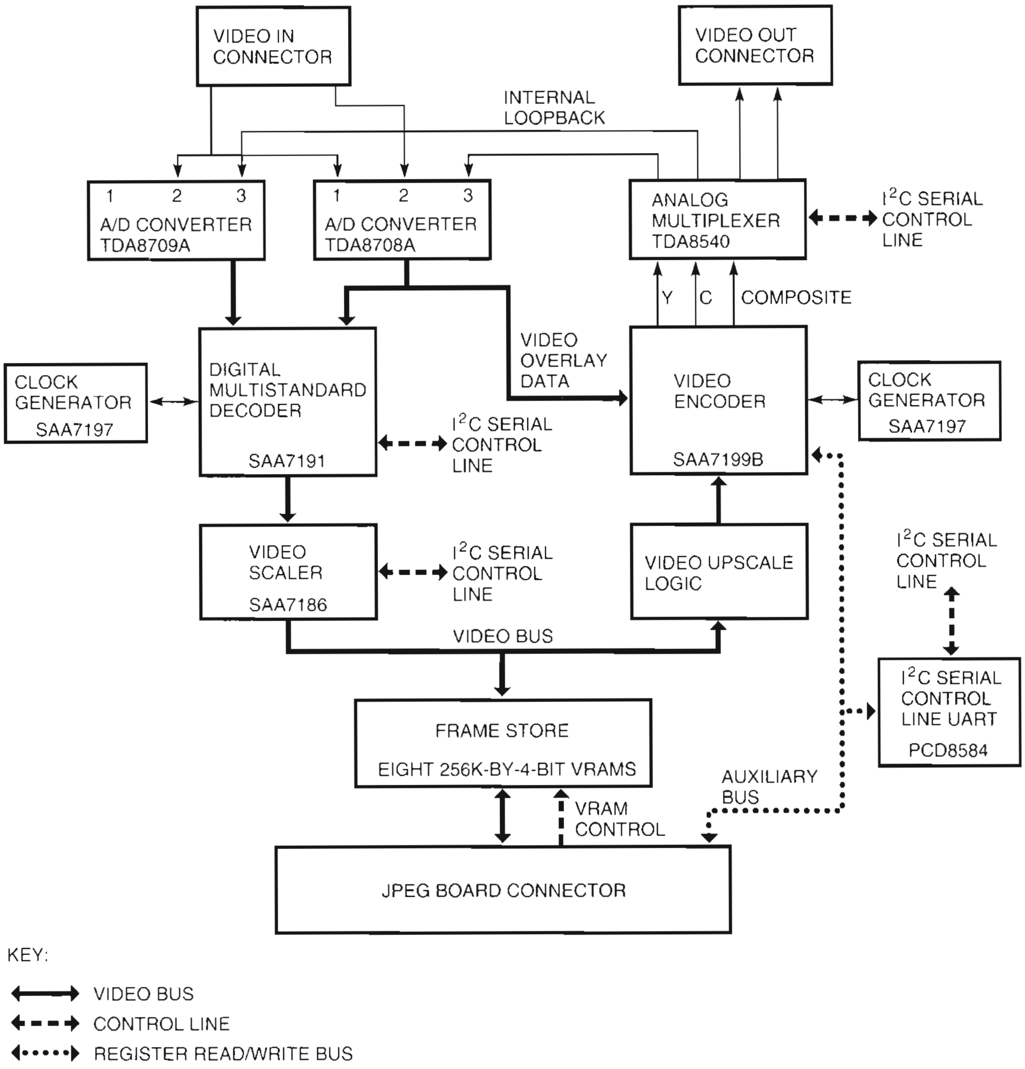


Figure 10
J300 Video I/O

set supports NTSC (M), PAL (B, G, H, D), and SECAM (B, G, H, D, K, K1) formats.[2] It also supports square pixels, where the sampling rate is changed to 12.272727 megahertz (MHz) for the NTSC format and to 14.75 MHz for the PAL and SECAM formats. In addition, the J300 uses the SAA7186, a digital video scaler chip that can scale the input to an arbitrary size and perform horizontal and vertical filtering.

The A/D converters digitize the incoming video signal to 256 levels. A video signal is composed of negative-going synchronization pulses, a color burst (to aid in decoding color information), and positive-going video.[11] As an aid to visualizing this, Figure 11 illustrates a simplified version of the drawing presented in the Color Television Studio Picture Line Amplifier Output Drawing.[11] The level before and after the synchronization pulses is referred to as blank level. Black level may or may not be the same as blank, depending on the standard. Video signals are 1 volt peak to peak.

The first stage included in the A/D converters is a three-to-one analog multiplexer. We used this circuit to allow two composite signals to be attached at the same time to support S-Video while allowing the third input to be used as an internal loop-back connection. The TDA8708A chip is used for composite video and for the luminance portion of S-Video. The TDA8709A chip is used only for the chrominance portion of S-Video.

The A/D converters contain an automatic gain control (AGC) circuit, which limits the A/D range. The bottom of the synchronization pulse is set at 0, and blank level is set at 64. Given these settings, peak white

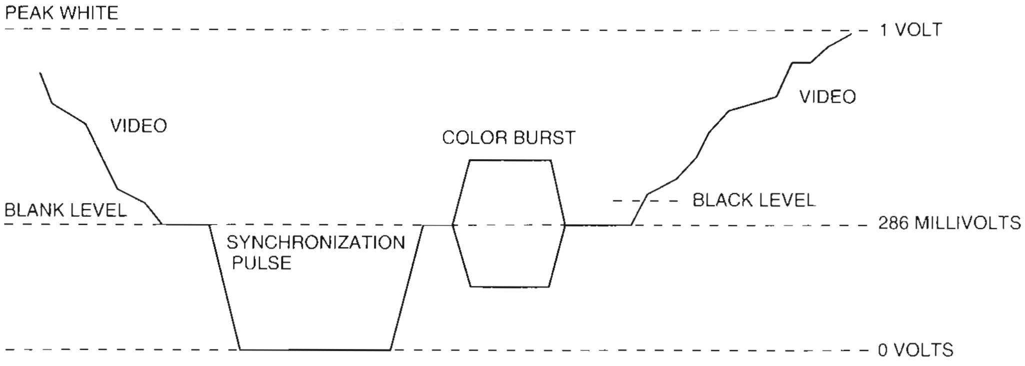


Figure 11
Depiction of Video Signal Terminology

corresponds to a value of 224. If the input video level tends to exceed 213, a peak white gain control loop is activated, which lowers the internal gain of the video. The SAA7191 processes the luminance, and the resulting range in the *Y* value is 16 for black and approximately 220 for white. As recommended by CCIR Report 601-2, there is room built into the two A/D converters and the DMSD to allow for additive noise that might be present in the distribution of video signals.[5]

The J300 video I/O design includes a video scaler so that the incoming video can be scaled down and filtered prior to compression. There are two primary reasons for this scaling. First, scaling reduces the amount of data to be processed, which results in a smaller compressed version of the image. Second, scaling removes any high-frequency noise in the image, which results in higher compression ratios. Unfortunately, if the user wishes to compress and also to view the incoming video stream, the video will more than likely be scaled again in the rendering circuit in the ASIC.

The J300 output video encoding circuit uses Philips' SAA7199B chip as the encoder. This component is followed by a low-pass filter and an analog multiplexer (Philips' TDA8540 chip), which functions as a 4 by 4 analog cross-point switch. The SAA7199B video encoder accepts digital data in a variety of formats, including 4:2:2 YUV. The SAA7199B processes the chrominance and luminance according to which standard is being encoded, either NTSC or PAL (B, G). The input range of the SAA7199B is compliant with CCIR Report 601-2 for YUV: *Y* varies from 16 to 235; *U* and *V* vary from 16 to 240. The analog multiplexer allows either the composite or S-Video output of the SAA7199B to be connected to the output connector. The switch also allows the video signals to be routed to the input circuit for an internal loop-back connection.

The J300 video I/O design initially included a frame store because the CL550 could not guarantee that compression of a field of video would be completed before the next field started. Even if the J300 scaled and filtered the video data prior to compression, some temporary storage was needed. We included eight 256K-by-4-bit VRAMs in the design for this storage.

In the mode where only the even field is being captured (which could be part of reducing the size of the final image from 640 by 480 pixels to 320 by 240 pixels), the J300 does not know when the system will request the next field of incoming video. VRAMs organized as 768 by 682 by 16 bits allow room to store two fields of either NTSC or PAL video. The incoming video stream continually alternates between these two buffers. The system then has the option of requesting the field that will provide the minimum input latency or the last complete field stored. Requesting the field with the minimum input latency creates the possibility that the compression and rendering operations will stall waiting for the finish of the video field being processed.

In another mode of operation, the memory is configured as a 1,024-by-512-by-16-bit buffer. This configuration is used when compressing or decompressing still images up to 1,024 pixels wide. Another use of the frame store organized in this way is for deinterlacing. In deinterlace mode, the even and odd fields are recombined to form one image. Deinterlacing allows capture of a full NTSC frame, but of only 512 lines of a PAL or SECAM frame. This restriction is due to the nature of the shift register cycles implemented in the VRAMs. A side effect of using this deinterlace mode when compressing the input is that the temporal effects of combining the two fields generate what the CL550 considers to be a large amount of high-spatial-frequency components in the image, thus resulting in poorer compression.

Audio Subsystem Design

The designers believed that the J300 design should include audio capabilities that complemented the video capabilities. Consequently, the design incorporates an analog codec (the CS4215 from Crystal Semiconductors) and a digital audio codec (the MC56401 from Motorola Semiconductors). These two chips provide all the audio I/O specified in the design. They communicate to the rest of the system by means of a serial digital interface.

To provide audio capabilities such as compression, decompression, and format conversion, the J300 includes a general-purpose DSP (DSP56001 from Motorola Semiconductors) with 8K by 24 bits of external RAM. This DSP can communicate to the audio codecs through an integrated port. It also handles the real-time nature of that interface by using a portion of the RAM to buffer the digital audio data.

The J300 offers the same type of DMA support for audio data as for video data. The audio interface controller ASIC, along with the DSP, provides support for four independent DMA streams. These streams correspond to the four possible sources or sinks of audio data: analog audio in, analog audio out, digital audio in, and digital audio out. The left channel of the analog audio connection can also be routed to the headphone/microphone jack. Figure 12 shows a block diagram of the audio portion of the J300.

Testability of I/O Sections

In the early stages of design, we were aware that built-in test features were needed to facilitate debugging and to reduce the amount of special audio- and video-specific test equipment required in manufacturing. Consequently, one J300 design goal was to include internal and external loop-back capability on all major I/O circuits. This goal was achieved with the exception of the digital audio circuit.

The video encoder can be programmed in test mode to output a flat field of red, green, or blue. This signal was used in internal and external loop-back. A comparison of the values obtained against known good values gives some level of confidence with regard to the video I/O stage. The designers accomplished external loop-back by using a standard S-Video cable.

The analog audio codec has internal loop-back capability, and a standard audio cable can be used for external loop-back tests. External loop-back tests of the headphone/microphone jack required a special adapter.

Even with this degree of internal and external loop-back capability, the goal was to be able to perform much more rigorous testing without the need of special instrumentation. Tests were developed that used two J300 systems to feed each other data. One J300 system output video data in NTSC or PAL formats of different test patterns, and the other J300 interpreted the signals. The designers used the same technique for both the digital and the analog audio codecs. This method provided a high degree of system coverage with no additional specialized test instruments.

Hardware Design Environment

The ASIC was designed completely in a hardware description language called Verilog, using no schematic sheets. At first, we simulated pieces of the design, building simple Verilog models for all the devices in the J300. We simulated complex chips such as the video scaler and the CL550 as data sources or sinks, reading data from or writing data to files in

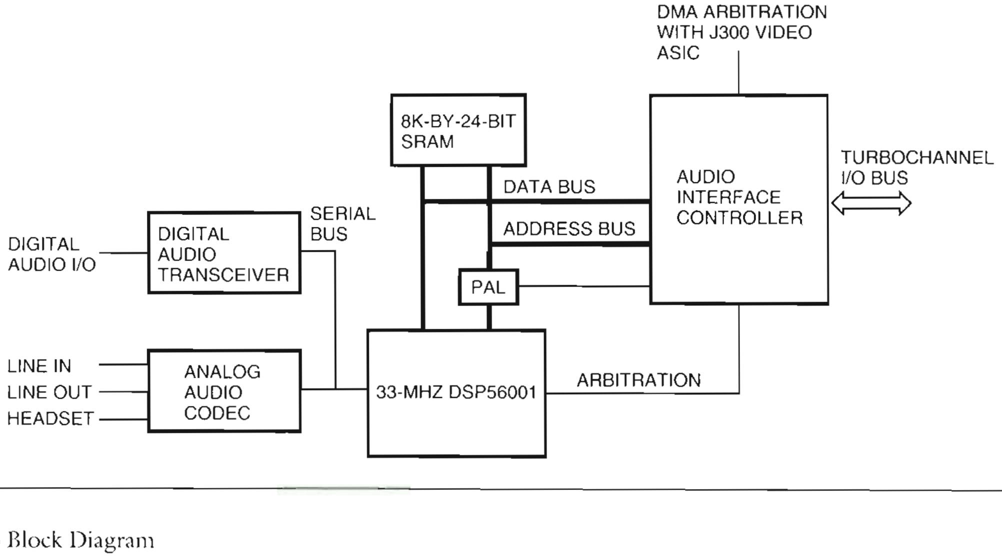


Figure 12
J300 Audio Block Diagram

memory. This approach limited the video data that could be compressed or decompressed to samples where both versions already existed. In all cases, the I/O ports on devices modeled included accurate timing information. Verilog includes the capability to incorporate user-defined routines written in the C programming language that can be compiled into a Verilog executable. The J300 design team took advantage of this capability by writing an interface that took TURBOchannel accesses from a portion of shared memory and used them to drive the Verilog model of the TURBOchannel bus. In that way, the designers could write test routines in C, compile them, and run them against the Verilog model of the ASIC and of the rest of the board design.

The Verilog model proved to be useful in developing manufacturing diagnostics and was used to some extent for driver and library code development. It was a very effective tool for the hardware designers, because much of the test code written during the design phase was used to bring up the hardware in the lab and later as example code for library development. Use of the Verilog model for software development was not as extensive as was hoped, however. The requirement to have a Verilog license available each time a model was invoked limited the number of users. There were enough licenses for hardware development, but few were left for software development. Another reason the software development team did not rely on using the Verilog model was that even though the model provided an accurate simulation of the hardware, the model was also very slow.

Concluding Remarks

With its Sound & Motion J300, FullVideo Supreme JPEG, and FullVideo Supreme products, Digital has achieved its goal of designing a hardware option that allows the integration of video into any workstation. The adapter performance on different platforms depends on many factors, chief among which are the efficiency of the bus design (either TURBOchannel or PCI), the amount of other traffic on the bus, and the design of the graphics device. As the performance of systems, particularly graphics devices, increases, the bottleneck in the J300 design becomes the pixel frequency through the J300 ASIC. For this reason, any future adapter designs should incorporate a higher pixel frequency.

The J300 family of products was the first to offer Digital's proprietary AccuVideo rendering technology, affording a high-quality yet low-cost solution for low-bit-depth frame buffers. Rendering video to 8 bits per pixel in combination with a high-speed bus allowed an architecture that is independent of the graphics subsystem.

Acknowledgments

The hardware engineering team included Rudy Stalzer, Petar Antonios, Tim Hellman, and Tom Fitzpatrick. Victor Bahl wrote the audio and video drivers, and Davis Pan wrote the code used by the DSP. Paul Gauthier contributed to the test routines available at power-up. Nagi Sivananjaiah wrote the diagnostics, and Lance Berc and Jim Gettys made major architectural contributions.

References

1. P. Bahl, "The J300 Family of Video and Audio Adapters: Software Architecture," *Digital Technical Journal,* vol. 7, no. 4 (1995, this issue): 34–51.

2. *Characteristics of Television Systems,* CCIR Report 624-2 (Geneva: International Radio Consultative Committee [CCIR], 1990).

3. *Encoding Parameters of Digital Television for Studios.* CCIR Report 601-2 (Geneva: International Radio Consultative Committee [CCIR], 1990).

4. *Information Technology—Digital Compression and Coding of Continuous-tone Still Images: Requirements and Guidelines,* ISO/IEC 10918-1:1994 (E) (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1994).

5. R. Ulichney, "Bresenham-style Scaling," *Proceedings of the IS&T Annual Conference* (Cambridge, Mass., 1993): 101–103.

6. R. Ulichney, *Digital Halftoning* (Cambridge, Mass.: The MIT Press, 1987).

7. 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.

8. L. Seiler and R. Ulichney, "Integrating Video Rendering into Graphics Accelerator Chips," *Digital Technical Journal.* vol. 7, no. 4 (1995, this issue): 76–88.

9. R. Ulichney, "Video Rendering," *Digital Technical Journal*, vol. 5, no. 2 (Spring 1993): 9–18.

10. R. Ulichney, "The Void-and-Cluster Method for Generating Dither Arrays," *IS&T/SPIE Symposium on Electronic Imaging Science and Technology,* San Jose, Calif., vol. 1913 (February 1993): 332–343.

11. *Industrial Electronics Tentative Standard No. 1, Color Television Studio Picture Line Amplifier Output Drawing* (Arlington, Va.: Industrial Electronics Association, November 1977). This publication is intended as a companion document to *Electrical Performance Standards—Monochrome Television Studio Facilities.* EIA-170 (Arlington, Va.: Industrial Electronics Association, November 1957).

Biographies

Kenneth W. Correll
After receiving a B.S.E.E. from the University of Washington in 1978, Ken Correll worked at Sperry Flight Systems for eight years, designing cockpit display systems. He joined Digital in 1986, focusing on the specification and design of advanced development projects concerned with the integration of live video and computer systems; he received a patent for some of this work. In 1990, Ken moved to Massachusetts and began work on the Jvideo and J300 projects. Since then he has been involved in the design of graphics ASICs in the Graphics and Multimedia Hardware Engineering Group within the Workstations Business Segment.

Robert A. Ulichney
Robert Ulichney received a Ph.D. from the Massachusetts Institute of Technology in electrical engineering and computer science and a B.S. in physics and computer science from the University of Dayton, Ohio. He joined Digital in 1978. He is currently a senior consulting engineer with Digital's Cambridge Research Laboratory, where he leads the Video and Image Processing project. He has filed several patents for contributions to Digital products in the areas of hardware and software-only motion video, graphics controllers, and hard copy. Bob is the author of *Digital Halftoning* and serves as a referee for a number of technical societies, including IEEE, of which he is a senior member.

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

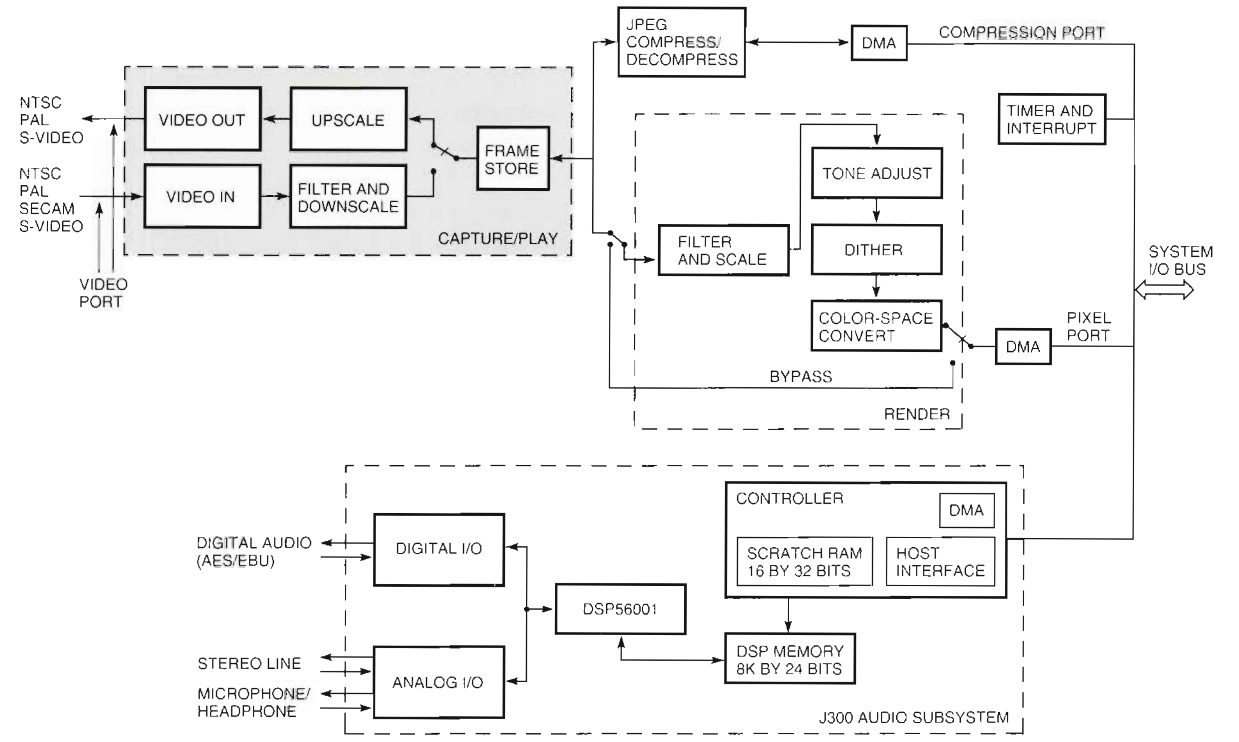


Figure 1
The Sound & Motion J300 Hardware Functional Diagram

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

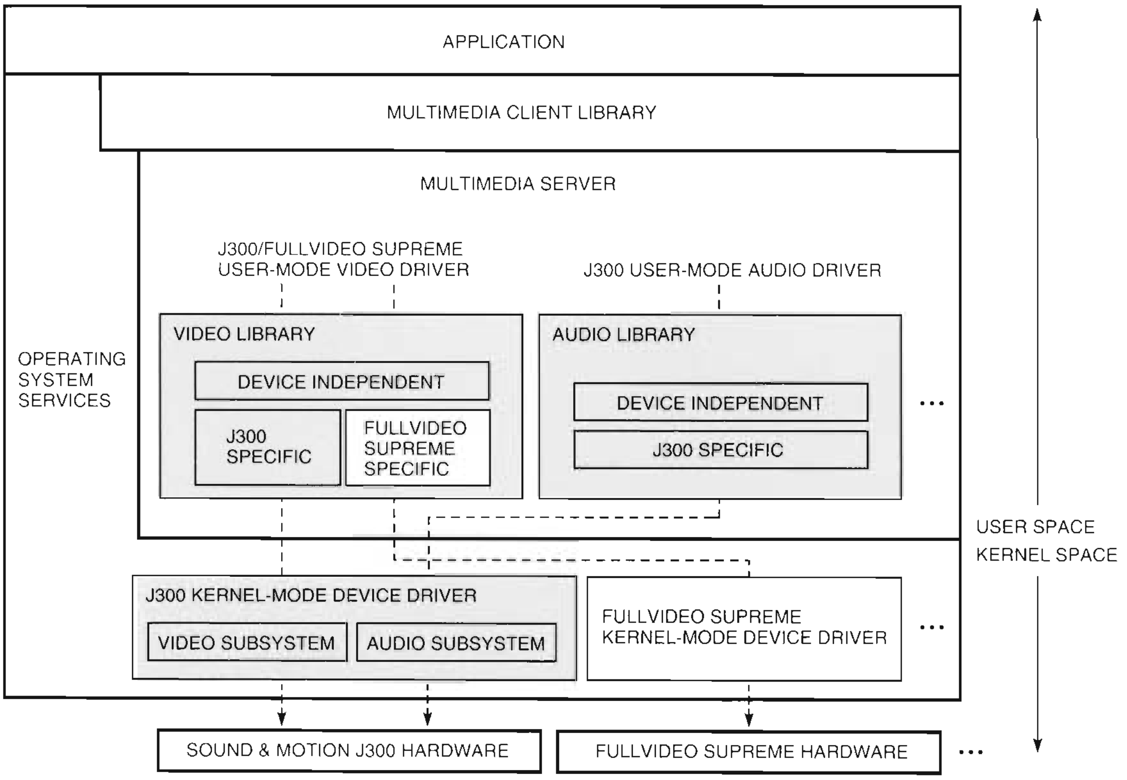

APPLICATION
MULTIMEDIA CLIENT LIBRARY
MULTIMEDIA SERVER
OPERATING SYSTEM SERVICES
J300/FULLVIDEO SUPREME USER-MODE VIDEO DRIVER
J300 USER-MODE AUDIO DRIVER
VIDEO LIBRARY
DEVICE INDEPENDENT
J300 SPECIFIC
FULLVIDEO SUPREME SPECIFIC
AUDIO LIBRARY
DEVICE INDEPENDENT
J300 SPECIFIC
...
USER SPACE
KERNEL SPACE
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

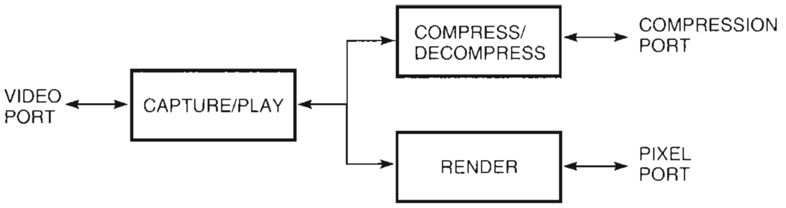
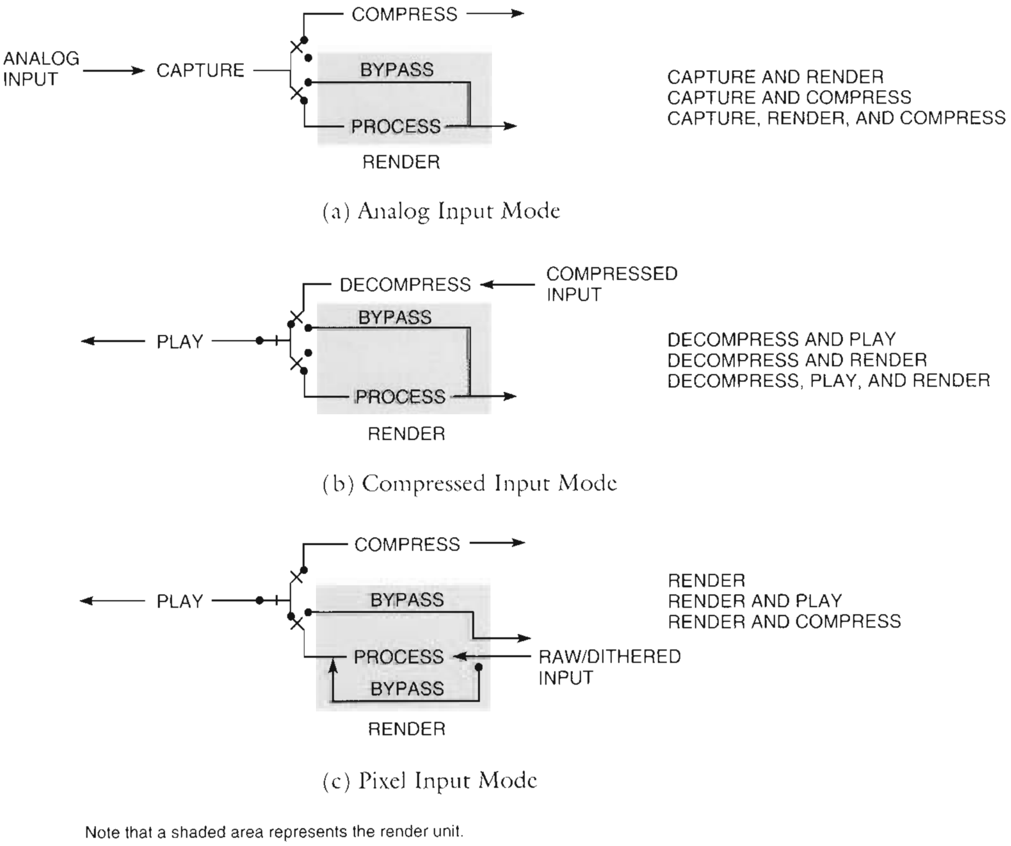


Figure 3
Conceptual Model for the J300 Video Subsystem Software

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.



(a) Analog Input Mode

(b) Compressed Input Mode

(c) Pixel Input Mode

Note that a shaded area represents the render unit.

Figure 4
The Nine Different J300 Video Flow Paths

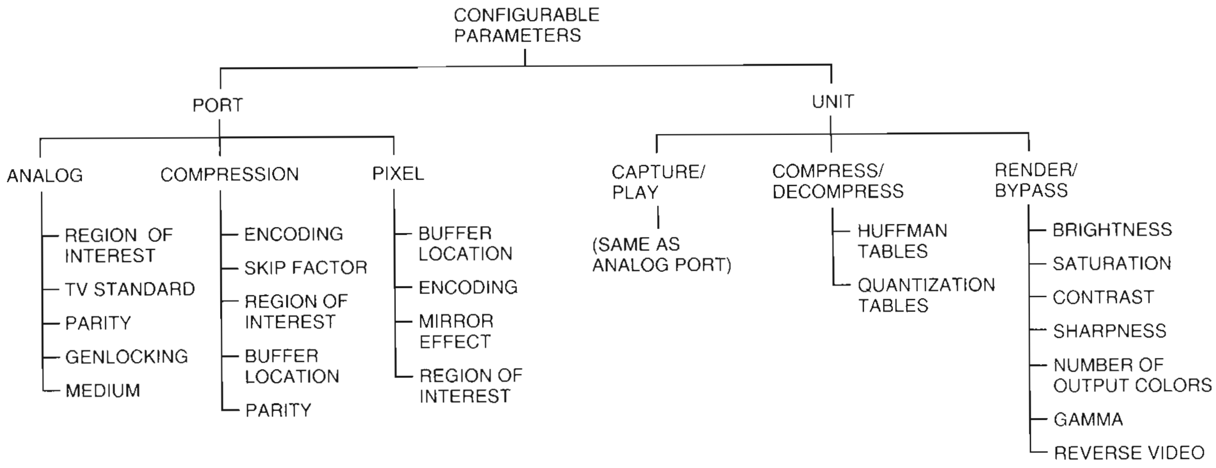


Figure 5
Tunable Parameters Provided by the J300 Video Library

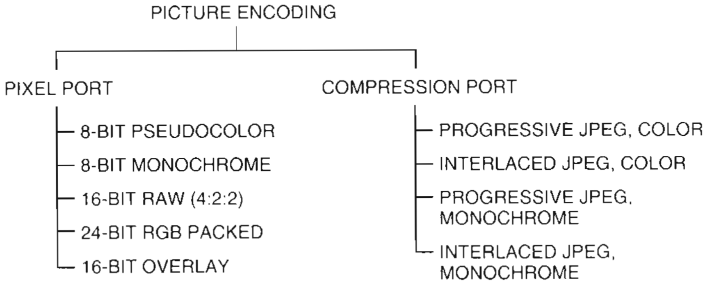


Figure 6
Valid Picture Encoding for the Two Digital I/O Ports

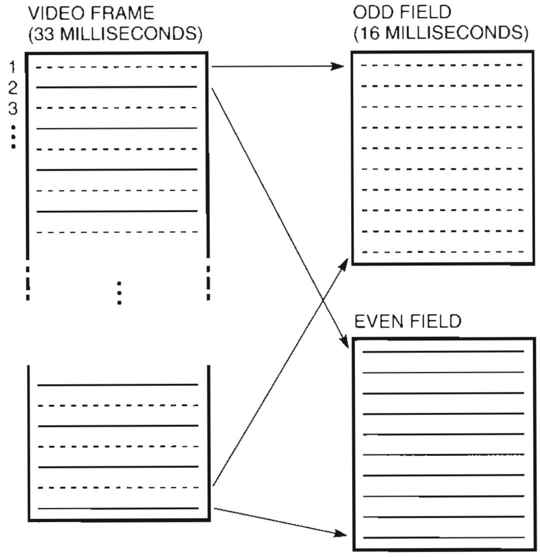


Figure 7
A Picture, Which May Be a Frame Or a Field

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 setup 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.

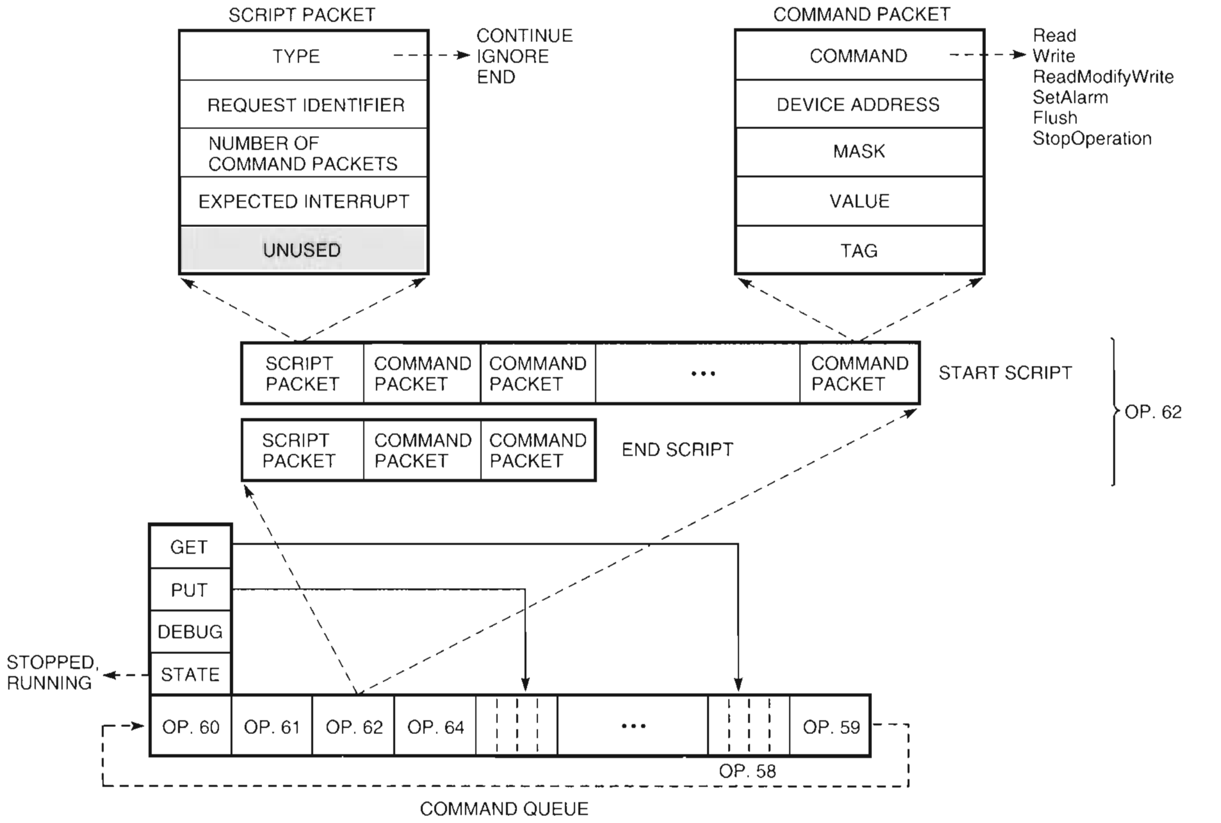


Figure 8
The Command Queue

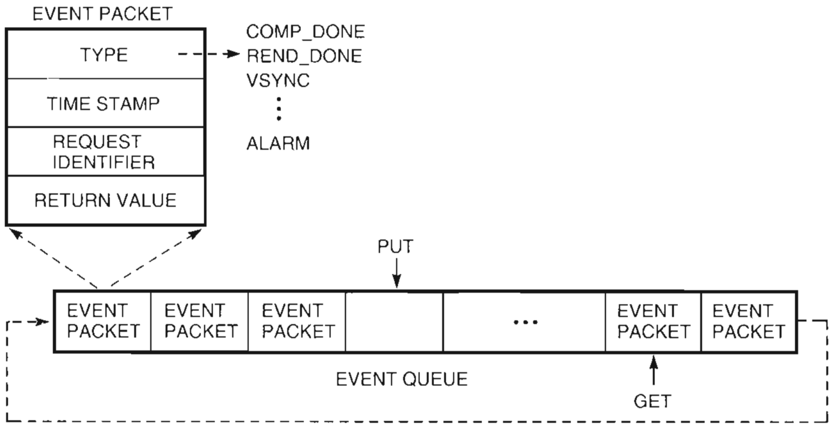
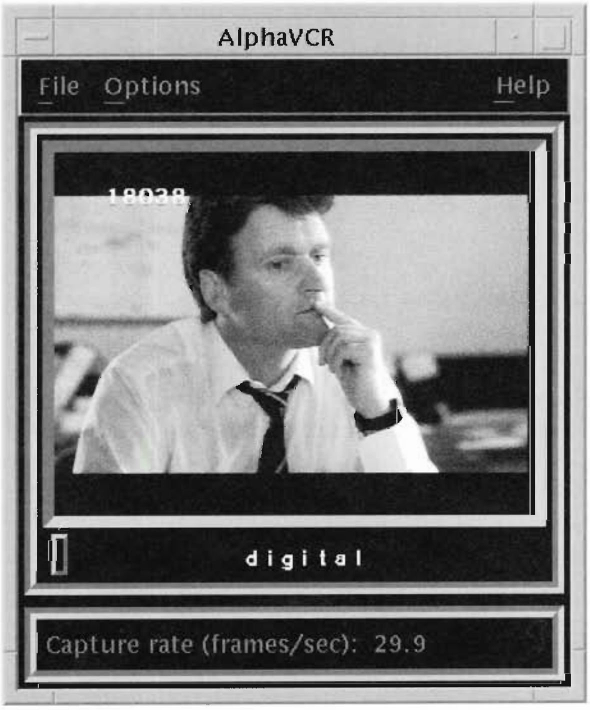


Figure 9
The Event Queue

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]
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



Capture rate (frames/sec): 29.9

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

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

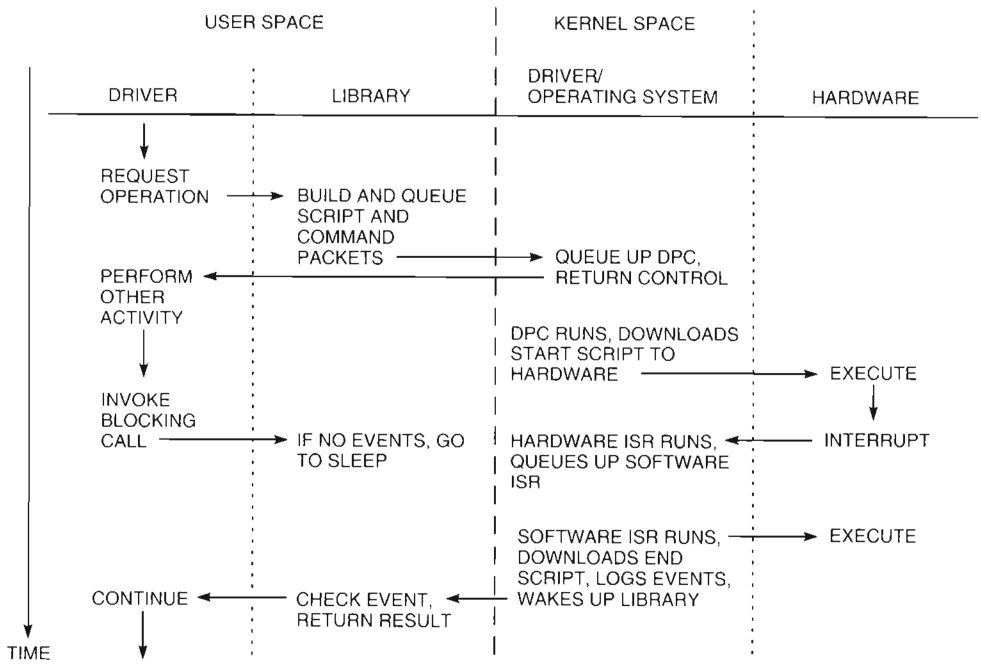


Figure 11
One Case of Simplified Flow Control When Using the Video Subsystem

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.

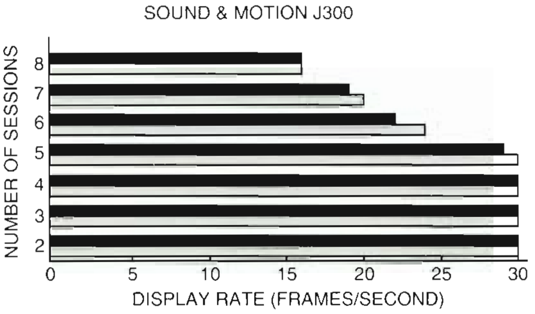
Video Subsystem Performance

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



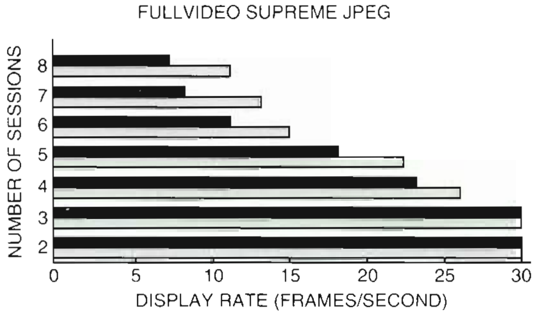
KEY:

□ 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

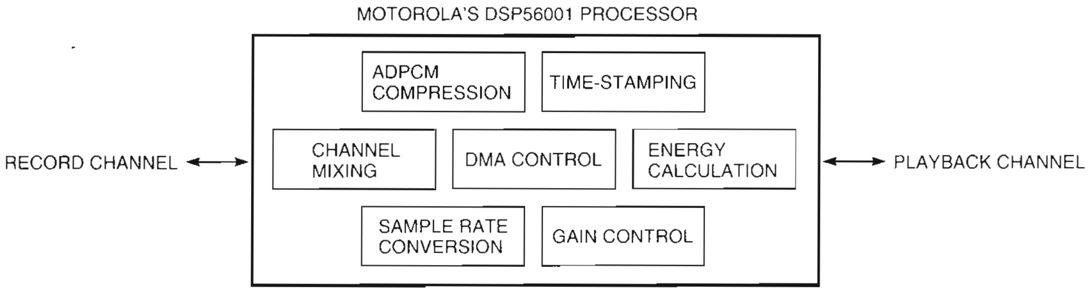


Figure 14
Some Audio Functions Supported by Motorola's DSP56001 Processor

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

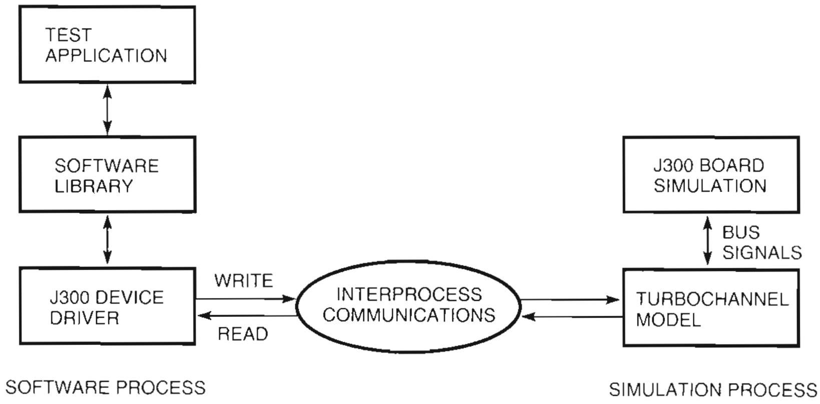


Figure 15
Hardware Simulation Environment for Software Development

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. Bolick 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.

Paramvir Bahl
Paul S. Gauthier
Robert A. Ulichney

Software-only Compression, Rendering, and Playback of Digital Video

Software-only digital video involves the compression, decompression, rendering, and display of digital video on general-purpose computers without specialized hardware. Today's faster processors are making software-only video an attractive, low-cost alternative to hardware solutions that rely on specialized compression boards and graphics accelerators. This paper describes the building blocks behind popular ISO, ITU-T, and industry-standard compression schemes, along with some novel algorithms for fast video rendering and presentation. A platform-independent software architecture that organizes the functionality of compressors and renderers into a unifying software interface is presented. This architecture has been successfully implemented on the Digital UNIX, the OpenVMS, and Microsoft's Windows NT operating systems. To maximize the performance of codecs and renderers, issues pertaining to flow control, optimal use of available resources, and optimizations at the algorithmic, operating-system, and processor levels are considered. The performance of these codecs on Alpha systems is evaluated, and the ensuing results validate the potential of software-only solutions. Finally, this paper provides a brief description of some sample applications built on top of the software architecture, including an innovative video screen saver and a software VCR capable of playing multiple, compressed bit streams.

Full-motion video is fast becoming commonplace to users of desktop computers. The rising expectations for low-cost, television-quality video with synchronized sound have been pushing manufacturers to create new, inexpensive, high-quality offerings. The bottlenecks that have been preventing the delivery of video without specialized hardware are being cast aside rapidly as faster processors, higher-bandwidth computer buses and networks, and larger and faster disk drives are being developed. As a consequence, considerable attention is currently being focused on efficient implementations of flexible and extensible software solutions to the problems of video management and delivery. This paper surveys the methods and architectures used in software-only digital video systems.

Due to the enormous amounts of data involved, compression is almost always used in the storage and transmission of video. The high level of information redundancy in video lends itself well to compression, and many methods have been developed to take advantage of this fact. While the literature is replete with compression methods, we focus on those that are recognized as standards, a requirement for open and interoperable systems. This paper describes the building blocks behind popular compression schemes of the International Organization for Standardization (ISO), the International Telecommunication Union-Telecommunication Standardization Sector (ITU-T), and within the industry.

Rendering is another enabling technology for video on the desktop. It is the process of scaling, color adjusting, quantization, and color space conversion of the video for final presentation on the display. As an example, Figure 1 shows a simple sequence of video decoding. In the section Video Presentation, we discuss rendering methods, along with some novel algorithms for fast video rendering and presentation, and describe an implementation that parallels the techniques used in Digital's hardware video offerings.

We follow that discussion with the section The Software Video Library, in which we present a common architecture for video compression, decompression, and playback that allows integration into Digital's multimedia products. We then describe two sample applications, the Video Odyssey screen saver

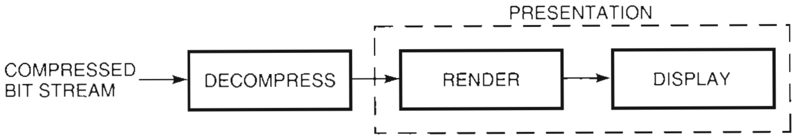


Figure 1
Components in a Video Decoder Pipeline

and a software-only video player. We conclude our paper by surveying related work in this rapidly evolving area of software digital video.

Video Compression Methods

A system that compresses and decompresses video, whether implemented in hardware or software, is called a video codec (for compressor/decompressor). Most video codecs consist of a sequence of components usually connected in pipeline fashion. The codec designer chooses specific components based on the design goals. By choosing the appropriate set of building blocks, a codec can be optimized for speed of decompression, reliability of transmission, better color reproduction, better edge retention, or to perform at a specific target bit rate. For example, a codec could be designed to trade off color quality for transmission bit rate by removing most of the color information in the data (color subsampling). Similarly a codec may include a simple decompression model (less processing per pixel) and a complex compression process to boost the playback rate at the expense of longer compression times. (Compression algorithms that take longer to compress than to decompress are said to be asymmetric.) Once the components and trade-offs have been chosen, the designer then fine tunes the codec to perform well in a specific application space such as teleconferencing or video browsing.

Video Codec Building Blocks

In this section, we present the various building blocks behind some popular and industry-standard video codecs. Knowledge of the following video codec components is essential for understanding the compression process and to appreciate the complexity of the algorithms.

Chrominance Subsampling Video is usually described as being composed of a sequence of images. Each image is a matrix of pixels, and each pixel is represented by three 8-bit values: a single luminance value (Y) that signifies brightness, and two chrominance values (U and V, or sometimes Cb and Cr) which, taken together, specify a unique color. By reducing the amount of color information in relation to luminance (subsampling the chrominance), we can reduce the size of an image with little or no perceptual effect. The most common chrominance subsampling technique decimates the color signal by 2:1 in the horizontal direction. This is done either by simply throwing out the color information of alternate pixels or by averaging the colors of two adjacent pixels and using the average for the color of the pixel pair. This technique is commonly referred to as 4:2:2 subsampling. When compared to a raw 24-bit image, this results in a compression of two-thirds. Decimating the color signal by 2:1 in both the horizontal and the vertical direction (by ignoring color information for alternate lines in the image) starts to result in some perceptible loss of color, but the compression increases to one-half. This is referred to as 4:2:0 subsampling: for every 4 luminance samples, there is a single color specified by a pair of chrominance values. The ultimate chrominance subsampling is to throw away all color information and keep only the luminance data (monochrome video). This not only reduces the size of the input data but also greatly simplifies processing for both the compressor and the decompressor, resulting in faster codec performance. Some teleconferencing systems allow the user to switch to monochrome mode to increase frame rate.

Transform Coding Converting a signal, video or otherwise, from one representation to another is the task of a transform coder. Transforms can be useful for video compression if they can convert the pixel data into a form in which redundant and insignificant information in the video's image can be isolated and removed. Many transforms convert the spatial (pixel) data into frequency coefficients that can then be selectively eliminated or quantized. Transform coders address three central issues in image coding: (1) decorrelation (converting statistically dependent image elements into independent spectral coefficients), (2) energy compaction (redistribution and localization of energy into a small number of coefficients), and (3) computational complexity. It is well documented that human vision is biased toward low frequencies. By transforming an image to the frequency domain, a codec can capitalize on this knowledge and remove or reduce the high-frequency components in the quantization step, effectively compressing the image. In addition, isolating and eliminating high-frequency components in an image results in noise reduction since most noise in video, introduced during the digitization step or from transmission interference, appears as high-frequency coefficients. Thus transforming helps compression by decorrelating (or whitening) signal samples and then discarding nonessential information from the image.

Unitary (or orthonormal) transforms fall into either of two classes: fixed or adaptive. Fixed transforms are independent of the input signal; adaptive transforms adapt to the input signal.[1] Examples of fixed transforms include the discrete Fourier transform (DFT), the discrete cosine transform (DCT), the discrete sine transform (DST), the Harr transform, and the Walsh-Hadamard transform (WHT). An example of an adaptive transform is the Karhunen-Loeve transform (KLT). Thus far, no transform has been found for pictorial information that completely removes statistical dependence between the transform coordinates. The KLT is optimum in the mean square error sense, and it achieves the best energy compaction; however, it is computationally very expensive. The WHT is the best in terms of computation cost since it requires only additions and subtractions; however, it performs poorly in decorrelation and energy compaction. A good compromise is the DCT, which is by far the most widely used transform in image coding. The DCT is closest to the KLT in the energy-packing sense, and, like the DFT, it has fast computation algorithms available for its implementation.[2] The DCT is usually applied in a sliding window on the image with a common window size of 8 pixels by 8 lines (or simply, 8 by 8). The window size (or block size) is important: if it is too small, the correlation between neighboring pixels is not exploited; if it is too large, block boundaries tend to become very visible. Transform coding is usually the most time-consuming step in the compression/decompression process.

Scalar Quantization A companion to transform coding in most video compression schemes is a scalar quantizer that maps a large number of input levels into a smaller number of output levels. Video is compressed by reducing the number of symbols that need to be encoded at the expense of reconstruction error. A quantizer acts as a control knob that trades off image quality for bit rate. A carefully designed quantizer provides high compression for a given quality. The simplest form of a scalar quantizer is a uniform quantizer in which the quantizer decision levels are of equal length or step size. Other important quantizers include Lloyd-Max's minimum mean square error (MMSE) quantizer and an entropy constraint quantizer.[3,4] Pulse code modulation (PCM) and adaptive differential pulse code modulation (ADPCM) are examples of two compression schemes that rely on pure quantization without regard to spatial and temporal redundancies and without exploiting the nonlinearity in the human visual system.

Predictive Coding Unless the image is changing rapidly, a video sequence will normally contain sequences of frames that are very similar. Predictive coding uses this fact to reduce the data volume by comparing pixels in the current frame with pixels in the same location in the previous frame and encoding the difference. A simple form of predictive coding uses the value of a pixel in one frame to predict the value of the pixel in the same location in the next frame. The prediction error, which is the difference between the predicted value and the actual value of the pixel, is usually small. Smaller numbers can be encoded using fewer quantization levels and fewer coding bits. Often the difference is zero, which can be encoded very compactly. Predictive coding can also be used within an image frame where the predicted value of a pixel may be the value of its neighbor or a weighted average of the pixels in the region. Predictive coding works best if the correlation between adjacent pixels that are spatially as well as temporally close to each other is strong. Differential PCM and delta modulation (DM) are examples of two compression schemes in which the predicted error is quantized and coded. The decompressor recovers the signal by applying this error to its predicted value for the sample. Lossless image compression is possible if the prediction error is coded without being quantized.

Vector Quantization An alternative to transform-based coding, vector quantization attempts to represent clusters of pixel data (vectors) in the spatial domain by predetermined codes.[5] At the encoder, each data vector is matched or approximated with a code word in the codebook, and the address or index of that code word is transmitted instead of the data vector itself. At the decoder, the index is mapped back to the code word, which is then used to represent the original data vector. Identical codebooks are needed at the compressor (transmitter) and the decompressor (receiver). The main complexity lies in the design of good representative codebooks and algorithms for finding best matches efficiently when exact matches are not available. Typically, vector quantization is applied to data that has already undergone predictive coding. The prediction error is mapped to a subset of values that are expected to occur most frequently. The process is called vector quantization because the values to be matched in the tables are usually vectors of two or more values. More elaborate vector quantization schemes are possible in which the difference data is searched for larger groups of commonly occurring values, and these groups are also mapped to single index values.

The amount of compression that results from vector quantization depends on how the values in the codebooks are calculated. Compression may be adjusted smoothly by designing a set of codebooks and picking the appropriate one for a given desired compression ratio.

Motion Estimation and Compensation Most codecs that use interframe compression use a more elaborate form of predictive coding than described above. Most videos contain scenes in which one or more objects move across the image against a fixed background or in which an object is stationary against a moving background. In both cases, many regions in a frame appear in the next frame but at different positions. Motion estimation tries to find similar regions in two frames and encodes the region in the second frame with a displacement vector (motion vector) that shows how the region has moved. The technique relies on the hypothesis that a change in pixel intensity from one frame to another is due only to translation.

For each region (or block) in the current frame, a displacement vector is evaluated by matching the information content of the measurement window with a corresponding measurement window W within a search area S, placed in the previous frame, and by searching for the spatial location that minimizes the matching criterion $\dot{d}$. Let $L_i(x,y)$ represent the pixel intensity at location (x,y) in frame i; and if (d_x,d_y) represents the region displacement vector for the interval $n(=(i+n)-i)$, then the matching criterion is defined as

$$\bar{d} = \min_{(d_x,d_y)\in S} \left\{ \sum_{(x,y)\in W} \left\| L_i(x,y) - L_{i\ n}(x-d_x, y-d_y) \right\| \right\} \quad n \geq 1 \qquad (1)$$

The most widely used distance measures are the absolute value $\|x\|=|x|$ and the quadratic norm $\|x\|=x^2$. Since finding the absolute minimum is guaranteed only by performing an exhaustive search of a series of discrete candidate displacements within a maximum displacement range, this process is computationally very expensive. A single displacement vector is assigned to all pixels within the region.

Motion compensation is the inverse process of using a motion vector to determine a region of the image to be used as a predictor.

Although the amount of compression resulting from motion estimation is large, the coding process is time-consuming. Fortunately, this time is needed only in the compression step. Decompression using motion estimation is relatively fast since no searching has to be done. For data replenishment, the decompressor simply uses the transmitted vector and accesses a region in the previous frame pointed to by the vector for data replenishment. Region size can vary among the codecs using motion estimation but is typically 16 by 16.

Frame/Block Skipping One technique for reducing data is to eliminate it entirely. In a teleconferencing situation, for example, if the scene does not change (above some threshold criteria), it may be acceptable to not send the new frame (drop or skip the frame). Alternatively, if bandwidth is limited and image quality is important, it may be necessary to drop frames to stay within a bit-rate budget. Most codecs used in teleconferencing applications have the ability of temporal subsampling and are able to gracefully degrade under limited bandwidth situations by dropping frames.

A second form of data elimination is spatial subsampling. The idea is similar to chrominance subsampling discussed previously. In most transform-based codecs, a block (8 by 8 or 16 by 16) is usually skipped if the difference between it and the previous block is below a predetermined threshold. The decompressor may reconstruct the missing pixels by using the previous block to predict the current block.

Entropy Encoding Entropy encoding is a form of statistical coding that provides lossless compression by coding input samples according to their frequency of occurrence. The two methods used most frequently include Huffman coding and run-length encoding.[6] Huffman coding assigns fewer bits to most frequently occurring symbols and more bits to the symbols that appear less often. Optimal Huffman tables can be generated if the source statistics are known. Calculating these statistics, however, slows down the compression process. Consequently, predeveloped tables that have been tested over a wide range of source images are used. A second and simpler method of entropy encoding is run-length encoding in which sequences of identical digits are replaced with the digit and the number in the sequence. Like motion estimation, entropy encoding puts a heavier burden on the compressor than the decompressor.

Before ending this section, we would like to mention that a number of other techniques, including object-based coding, model-based coding, segmentation-based coding, contour-texture oriented coding, fractal coding, and wavelet coding are also available to the codec designer. Thus far, our coverage has concentrated on explaining only those techniques that have been used in the video compression schemes currently supported by Digital. In the next section, we describe some hybrid schemes that employ a number of the techniques described above; these schemes are the basis of several international video coding standards.

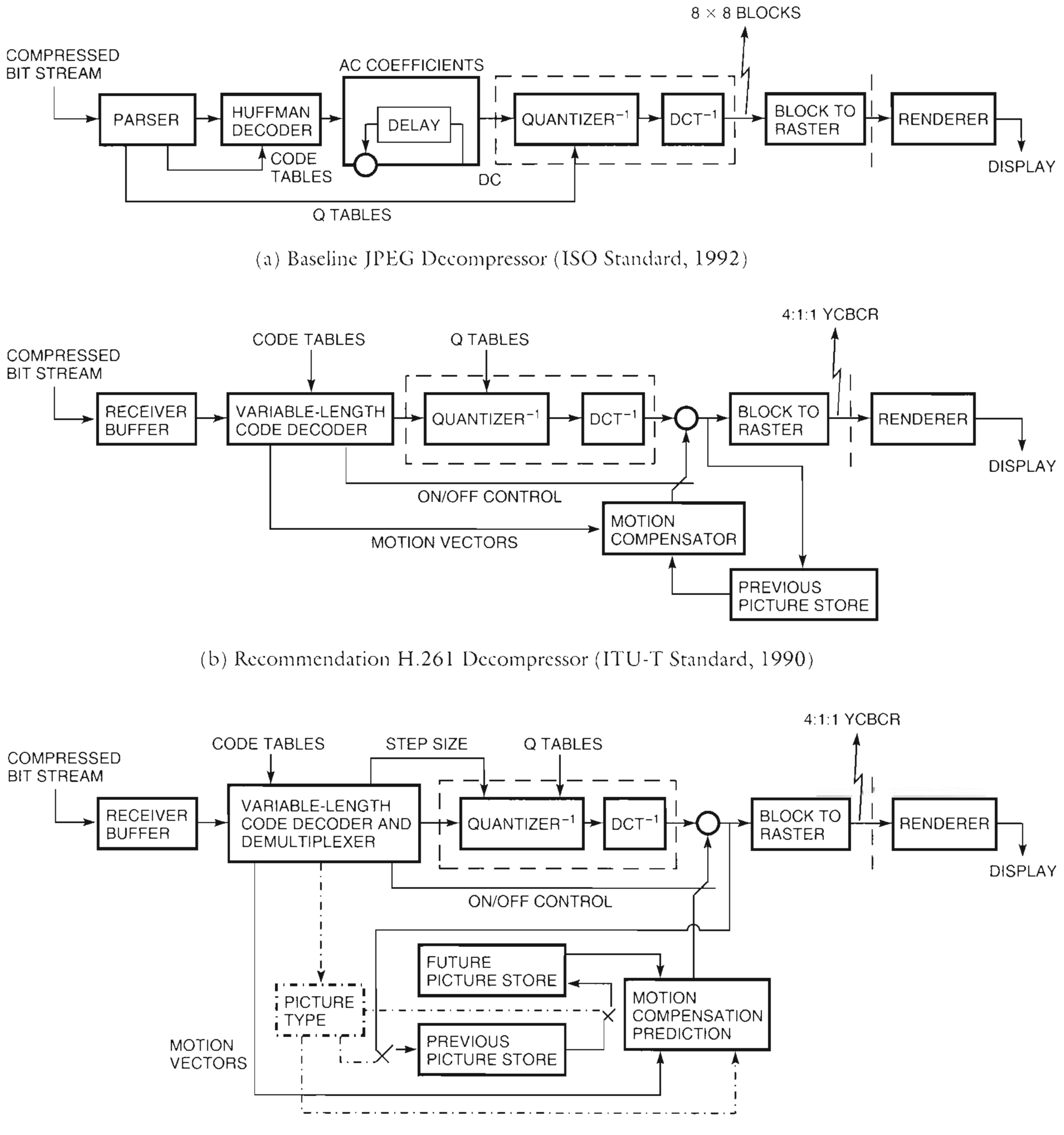
Overview of Popular Video Compression Schemes

The compression schemes presented in this section can be collectively classified as first-generation video coding schemes.[7] The common assumption in all these methods is that there is statistical correlation between pixels. Each of these methods attempts to exploit this correlation by employing redundancy reduction techniques to achieve compression.

Motion-JPEG Algorithm Motion-JPEG (or M-JPEG) compresses each frame of a video sequence using the ISO's Joint Photographic Experts Group (JPEG) continuous-tone, still-image compression standard.[8] As such, it is an intraframe compression scheme. It is not wed to any particular subsampling format, image color space, or image dimensions, but most typically 4:2:2 subsampled YCbCr, source input format (SIF, 352 by 240) data is used. The JPEG standard specifies both lossy and lossless compression schemes. For video, only the lossy baseline DCT coding scheme has gained acceptance. The scheme relies on selective quantization of the frequency coefficients followed by Huffman and run-length encoding for its compression. The standard defines a bit-stream format that contains both the compressed data stream and coding parameters such as the number of components, quantization tables, Huffman tables, and sampling factors. Popular M-JPEG file formats usually build on top of the JPEG-specified formats with little or no modification. For example, Microsoft's audio-video interleaved (AVI) format encapsulates each JPEG frame with its associated audio and adds an index to the start of each frame at the end of the file. Video editing on a frame-by-frame basis is possible with this format. Another advantage is frame-limited error propagation in networked, distributed applications. Many video digitizer boards incorporate JPEG compression in hardware to compress and decompress video in real time. Digital's Sound & Motion J300 and FullVideo Supreme JPEG are two such boards.[9,10] The baseline JPEG codec is a symmetric algorithm as may be seen in Figure 2a and Figure 3.

ITU-T's Recommendation H.261 The ITU-T's Recommendation H.261 is a motion-compensated, DCT-based video coding standard.[11] Designed for the teleconferencing market and developed primarily for low-bit-rate Integrated Services Digital Network (ISDN) services, H.261 shares similarities with ISO's JPEG still-image compression standard. The target bit rate is $p \times 64$ kilobits per second with p ranging between 1 and 30 (H.261 is also known as $p \times 64$). Only two frame resolutions, common intermediate format (CIF, 352 by 288) and quarter-CIF (QCIF, 176 by 144), are allowed. All standard-compliant codecs must be able to operate with QCIF; CIF is optional. The input color space is fixed by the International Radio Consultative Committee (CCIR) 601 YCbCr standard's with 4:2:0 subsampling (subsampling of chrominance components by 2:1 in both the horizontal and the vertical direction). Two types of frames are defined: key frames that are coded independently and non–key frames that are coded with respect to a previous frame. Key frames are coded in a manner similar to JPEG. For non–key frames, block-based motion compensation is performed to compute interframe differences, which are then DCT coded and quantized. The block size is 16 by 16, and each block can have a different quantization table. Finally, a variable word-length encoder (usually employing Huffman and run-length methods) is used for coding the quantized coefficients. Rate control is done by dropping frames, skipping blocks, and increasing quantization. Error correction codes are embedded in the bit stream to help detect and possibly correct transmission errors. Figure 2b shows a block diagram of an H.261 decompressor.

ISO's MPEG-1 Video Standard The MPEG-1 video standard was developed by ISO's Motion Picture Experts Group (MPEG). Like the H.261 algorithm, MPEG-1 is also an interframe video codec that removes spatial redundancy by compressing key frames using techniques similar to JPEG and removes temporal redundancy through motion estimation and compensation.[11,12] The standard defines three different types of frames or pictures: intra or I-frames that are compressed independently; predictive or P-frames that use motion compensation from the previous I- or P-frame; and bidirectional or B-frames that contain blocks predicted from either a preceding or following P- or I-frame (or interpolated from both). Compression is greatest for B-frames and least for I-frames. (A fourth type of frame, called the D-frame or the DC-intracoded frame, is also defined for improving fast-forward-type access, but it is hardly ever used.) There is no restriction on the input frame dimensions, though the target bit rate of 1.5 megabits per second is for video containing SIF frames. Subsampling is fixed at 4:2:0. MPEG-1 employs adaptive quantization of DCT coefficients for compressing I-frames and for compressing the difference between actual and predicted blocks in P- and B-frames. A 16-by-16 sliding window, called a macroblock, is used in motion estimation; and a variable word-length encoder is used in the final step to further lower the output bit rate. The full MPEG-1 standard specifies a system stream that includes a video and an audio substream, along with timing information needed for synchronization between the two. The video substream contains the compressed video data and coding parameters such as picture rate, bit rate, and image size. MPEG-1 has become increasingly popular primarily because it offers better compression than JPEG without compromising on quality. Several vendors and chip manufacturers offer specialized hardware for MPEG compression and decompression. Figure 2c shows a block diagram of an MPEG-1 video decompressor.



(a) Baseline JPEG Decompressor (ISO Standard, 1992)

(b) Recommendation H.261 Decompressor (ITU-T Standard, 1990)

(c) MPEG-1 Video Decompressor (ISO Standard, 1994)

Figure 2
Playback Configurations for Compressed Video Streams

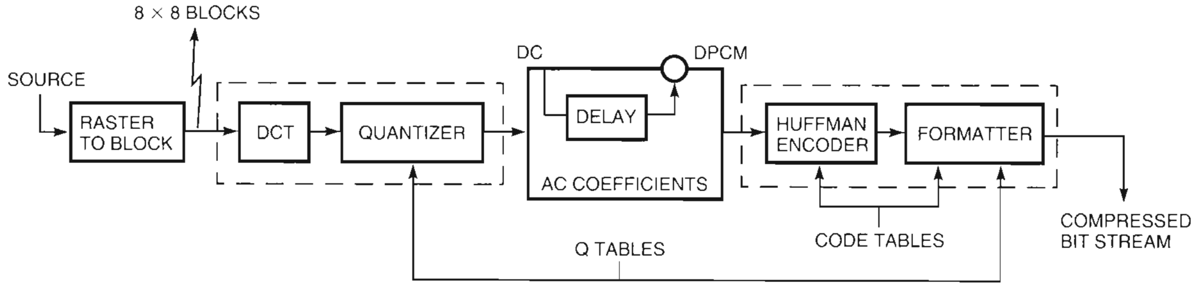


Figure 3
ISO's Baseline JPEG Compressor

Intel's INDEO Video Compression Algorithm Intel's proprietary INDEO video compression algorithm is used primarily for video presentations on personal computer (PC) desktops. It employs color subsampling, pixel differencing, run-length encoding, vector quantization, and variable word-length encoding. The chrominance components are heavily subsampled. For every block of 4-by-4 luminance samples, there is a single sample of Cb and Cr. Furthermore, samples are shifted one bit to convert them to 7-bit values. The resulting precompression format is called YVU9, because on average there are 9 bits per pixel. This subsampling alone yields a reduction of 9/24. Run-length encoding is employed to encode any run of zero pixel differences.

PCWG's INDEO-C Video Compression Algorithm INDEO-C is the video compression component of a teleconferencing system derived from the Personal Conferencing Specification developed by the Personal Conferencing Work Group (PCWG), an industry group led by Intel Corporation. Like the MPEG standard, the PCWG specification defines the compressed bit stream and the decoder but not the encoder. INDEO-C is optimized for low-bit-rate, ISDN-based connections and, unlike its desktop compression cousin, is transform-based. It is an interframe algorithm that uses motion estimation and a 4:1 chrominance subsampling in both directions. Spatial and temporal loop filters are used to remove high-frequency artifacts. The transform used for converting spatial data to frequency coefficients is the slant transform, which has the advantage of requiring only shifts and adds with no multiplies. Like the DCT, the fast slant transform (FST) is applied on image subblocks for coding both intraframes and difference frames. As was the case in other codecs, run-length coding and Huffman coding are employed in the final step. Compression and decompression of video in software is faster than other interframe schemes like MPEG-1 and H.261.

Compression Schemes under Development In addition to the five compression schemes described in this section, four other video compression standards, which are currently in various stages of development within ISO and ITU-T, are worth mentioning: ISO's MPEG-2, ITU-T's Recommendation H.262, ITU-T's Recommendation H.263, and ISO's MPEG-4.[13,14] Although the techniques employed in MPEG-2, H.262, and H.263 compression schemes are similar to the ones discussed above, the target applications are different. H.263 focuses on providing low-bit-rate video (below 64 kilobits per second) that can be transmitted over narrowband channels and used for real-time conversational services. The codec would be employed over the plain old telephone system (POTS) with modems that have the V.32 and the V.34 modem technologies. MPEG-2, on the other hand, is aimed at bit rates above 2 megabits per second, which support a wide variety of formats for multimedia applications that require better quality than MPEG-1 can achieve. One of the more popular target applications for MPEG-2 is coding for high-definition television (HDTV). It is expected that ITU-T will adapt MPEG-2 so that Recommendation H.262 will be very similar, if not identical, to it. Finally, like Recommendation H.263, ISO's MPEG-4's charter is to develop a generic video coding algorithm for low-bit-rate multimedia applications over a public switched telephone network (PSTN). A wide variety of applications, including those operating over error-prone radio channels, are being targeted. The standard is expected to embrace coding methods that are very different from its precursors and will include the so-called second-generation coding techniques.[7] MPEG-4 is expected to reach draft stage by November 1997.

This ends our discussion on video compression techniques and standards. In the next section, we turn our attention to the other component of the video playback solution, namely video rendering. We describe the general process of video rendering and present a novel algorithm for efficient mapping of out-of-range colors to feasible red, green, and blue (RGB) values that can be represented on the target display device. Out-of-range colors can occur when the display quality is adjusted during video playback.

Video Presentation

Video presentation or rendering is the second important component in the video playback pipeline (see Figure 1). The job of this subsystem is to accept decompressed video data and present it in a window of specified size on the display device using a specified number of colors. The basic components are sketched in Figure 4 and described in more detail in a previous issue of this *Journal*.[15] Today, most desktop systems do not include hardware options to perform these steps, but some interesting cases are available as described in this issue.[9,16] When such accelerators are not available, software-only implementation is necessary. Software

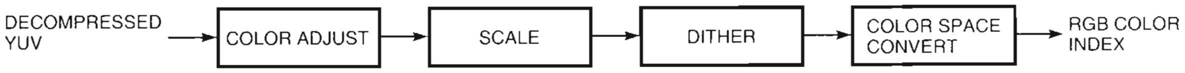


Figure 4
Components of Video Rendering

rendering algorithms, although very efficient, can still consume as many computation cycles as are used to decompress the data.

All major video standards represent image data in a luminance-chrominance color space. In this scheme, each pixel is composed of a single luminance component, denoted as Y, and two chrominance components that are sometimes referred to as color difference signals Cb and Cr, or signals U and V. The relationship between the familiar RGB color space and YUV can be described by a 3-by-3 linear transformation:

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \mathbf{M} \begin{bmatrix} y \\ u \\ v \end{bmatrix}, \qquad (2)$$

where the transformation matrix,

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & a \\ 1 & b & c \\ 1 & d & 0 \end{bmatrix}. \qquad (3)$$

The matrix is somewhat simple with only four values that are not 0 or 1. These constants are $a = 1.402$, $b = -.344$, $c = -.714$, and $d = 1.722$.

The RGB color space cube becomes a parallelepiped in YUV space. This is pictured in Figure 5, where the black corner is at the bottom, and the white corner is at the top; the red, green, and blue corners are as labeled. The chrominance signals U and V are usually subsampled, so the rendering subsystem must first restore these components and then transform the YUV triplets to RGB values.

Typical frame buffers are configured with 8 bits of color depth. This hardware colormap must, in general, be shared by multiple applications, which puts a premium on each of the 256 color slots in the map. Each application, therefore, must be able to request rendering to a limited number of colors. This can be accomplished most effectively with a multilevel dithering scheme, as represented by the dither block in Figure 4.

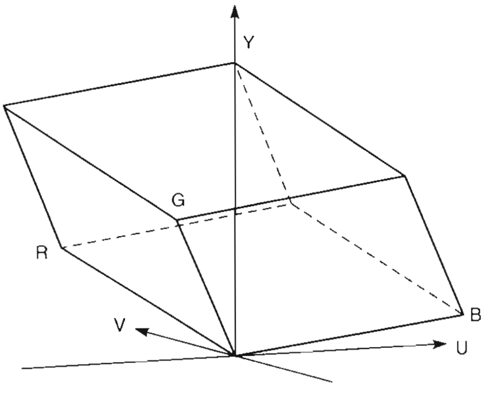


Figure 5
The RGB "Cube" in YUV Space

The color adjustment block controls brightness, contrast and saturation by means of simple look-up tables.

Along with up-sampling the chrominance, the scale block in Figure 4 can also change the size of the image. Although arbitrary scaling is best performed in combination with filtering, it is found to be too expensive to do in a software-only implementation. For the case of enlargement, a trade-off can be made between image quality and speed; contrary to what is shown in Figure 4, image enlargement can occur after dithering and color space converting. Of course, this would result in scaled dithered pixels, which are certainly less desirable, but it would also result in faster processing.

To optimize computational efficiency, color space conversion from YUV to RGB takes place after YUV dithering. Dithering greatly reduces the number of YUV triplets, thus allowing a single look-up table to perform the color space conversion to RGB as well as map to the final 8-bit color index required by the graphics display system. Digital pioneered this idea and has used it in a number of hardware and software-only products.[17]

Mapping Out-of-Range Colors

Besides the obvious advantages of speed and simplicity, using a look-up table to convert dithered YUV values to RGB values has the added feature of allowing careful mapping of out-of-range YUV values. Referring again to Figure 5, the RGB solid describes those r, g, and b values that are feasible, that is, have the normalized range $0 \le r, g, b \le 1$. The range of possible values in YUV space are those for $0 \le y \le 1$ and $-.5 \le u, v \le .5$. It turns out that the RGB solid occupies only 23.3 percent of this possible YUV space; thus there is ample possibility for so-called infeasible or out-of-range colors to occur. Truncating the r, g, and b values of these colors has the effect of mapping back to the RGB parallelepiped along lines perpendicular to its nearest surface; this is undesirable since it will result in changing both the hue angle or polar orientation in the chrominance plane and the luminance value. By storing the mapping in a look-up table, decisions can be made a priori as to exactly what values the out-of-range values should map to.

There is a mapping where both the luminance or y value and the hue angle are held constant at the expense of a change in saturation. This section details how a closed-form solution can be found for such a mapping. Figure 6 is a cross section of the volume in Figure 5 through a plane at $y = y_0$. The object is to find the point on the surface of the RGB parallelepiped that maps the out-of-range point (y_0, u_0, v_0) in the plane of constant y_0 (constant luminance) and along a straight line to the u-v origin (constant hue angle). The solution is the intersection of the closest RGB surface and the line between (y_0, u_0, v_0) and $(y_0, 0, 0)$. This line can

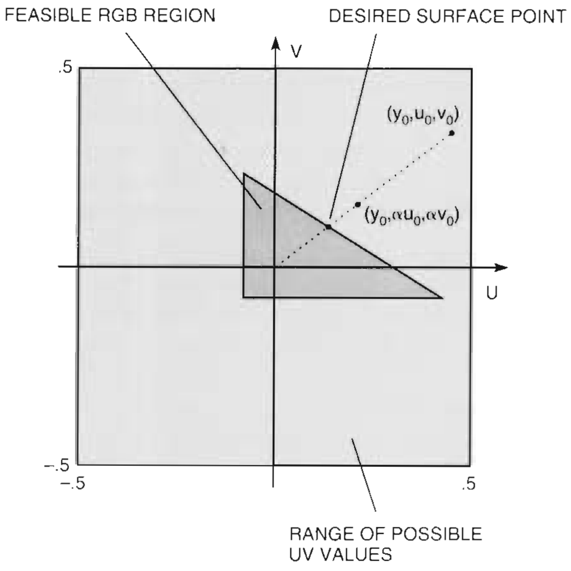


Figure 6
Mapping Out-of-Range YUV Points to the Surface of the RGB Parallelepiped in a Plane of Constant y_0

be parametrically represented as the locus $(y_0, \alpha u_0, \alpha v_0)$ for a single parameter α. The RGB values for these points are

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \mathbf{M} \begin{bmatrix} y_0 \\ \alpha u_0 \\ \alpha v_o \end{bmatrix} = \begin{bmatrix} \alpha(av_0)+y_0 \\ \alpha(bu_0+cv_0)+y_0 \\ \alpha(du_0)+y_0 \end{bmatrix}, \qquad (4)$$

where the matrix **M** is as given in equation (2). To find where this parametric line will intersect the RGB parallelepiped, we can first solve for the α at the intercept values at each of the six bounding surface planes as follows:

Surface Plane	Intercept Value
$r=1$	$\alpha_1 = (1-y_0)/av_0$
$g=1$	$\alpha_2 = (1-y_0)/(bu_0+cv_0)$
$b=1$	$\alpha_3 = (1-y_0)/du_0$
$r=0$	$\alpha_4 = (\alpha_1-1)$
$g=0$	$\alpha_5 = (\alpha_2-1)$
$b=0$	$\alpha_6 = (\alpha_3-1)$

Exactly three α_i will be negative, with each describing the intercept with extended RGB surface planes opposite the u-v origin. Of the remaining three α_i, the two largest values will describe intercepts with extended RGB surface planes in infeasible RGB space. This is because the RGB volume, a parallelepiped, is a convex polyhedron. Thus the solution must simply be the smallest positive α_i. Plugging this value of α into equation (4) produces the desired RGB value.

The Software Video Library

When we started this project, we had two objectives in mind: to showcase the processing power of Digital's newly developed Alpha processor and to use this power to make digital video easily available to developers and end users by providing extremely low-cost solutions. We knew that because of the compute-intensive nature of video processing, Digital's Alpha processor would outperform any competitive processor in a head-to-head match. By providing the ability to manipulate good-quality desktop video without the need for additional hardware, we wanted to make Alpha-based systems the computers of choice for end users who wanted to incorporate multimedia into their applications.

Our objectives translated to the creation of a software video library that became a reality because of three key observations. The first one is embedded in our motivation: processors had become powerful enough to perform complex signal-processing operations at real-time rates. With the potential of even greater speeds in the near future, low-cost multimedia solutions would be possible since audio and video decompression could be done on the native processor without any additional hardware.

A second observation was that multiple emerging audio/video compression standards, both formal and industry de facto, were gaining popularity with application vendors and hence needed to be supported on Digital's platforms. On careful examination of the compression algorithms, we observed that most of the prominent schemes used common building blocks (see Figure 2). For example, all five international standards—JPEG, MPEG-1, MPEG-2, H.261, and H.263—have DCT-based transform coders followed by a quantizer. Similarly, all five use Huffman coding in their final step. This meant that work done on one codec could be reused for others.

A third observation was that the most common component of video-based applications was video playback (for example, videoconferencing, video-on-demand, video player, and desktop television). The output decompressed streams from the various decoders have to be software-rendered for display on systems that do not have support for color space conversion and dithering in their graphics adapters. An efficient software rendering scheme could thus be shared by all video players.

With these observations in mind, we developed a software video library containing quality implementations of ISO, ITU-T, and industry de facto video coding standards. In the sections to follow, we present the architecture, implementation, optimization, and performance of the software video library. We complete our presentation by describing examples of video-based applications written on top of this library, including a novel video screen saver we call Video Odyssey and a software-only video player.

Architecture

Keeping in mind the observations outlined above, we designed a software video library (SLIB) that would

- Provide a common architecture under which multiple audio and video codecs and renderers could be accessed
- Be the lowest, functionally complete layer in the software video codec hierarchy
- Be fast, extensible, and thread-safe, providing reentrant code with minimal overhead
- Provide an intuitive, simple, flexible, and extensible application programming interface (API) that supports a client-server model of multimedia computing
- Provide an API that would accommodate multiple upper layers, allowing for easy and seamless integration into Digital's multimedia products

Our intention was not to create a library that would be exposed to end-user applications but to create one that would provide a common architecture for video codecs for easy integration into Digital's multimedia products. SLIB's API was purposely designed to be a superset of Digital's Multimedia Services' API for greater flexibility in terms of algorithmic tuning and control. The library would fit well under the actual programming interface provided to end users by Digital's Multimedia Services. Digital's Multimedia API is the same as Microsoft's Video For Windows API, which facilitates the porting of multimedia applications from Windows and Windows NT to Digital UNIX and OpenVMS platforms. Figure 7 shows SLIB in relation to Digital's multimedia software hierarchy. The shaded regions indicate the topics discussed in this paper.

As mentioned, the library contains routines for audio and video codecs and Digital's propriety video-rendering algorithms. The routines are optimized both algorithmically and for the particular platform on which they are offered. The software has been successfully implemented on multiple platforms, including the Digital UNIX, the OpenVMS, and Microsoft's Windows NT operating systems.

Three classes of routines are provided for the three subsystems: (1) video compression and decompression, (2) video rendering, and (3) audio processing. For each subsystem, routines can be further classified as (a) setup routines, (b) action routines, (c) query routines, and (d) teardown routines. Setup routines create and initialize all relevant internal data structures. They also compute values for the various look-up tables such as the ones used by the rendering subsystem. Action routines perform the actual coding, decoding, and rendering operations. Query routines may be used before setup or between action routines. These provide the programmer with information about the capability

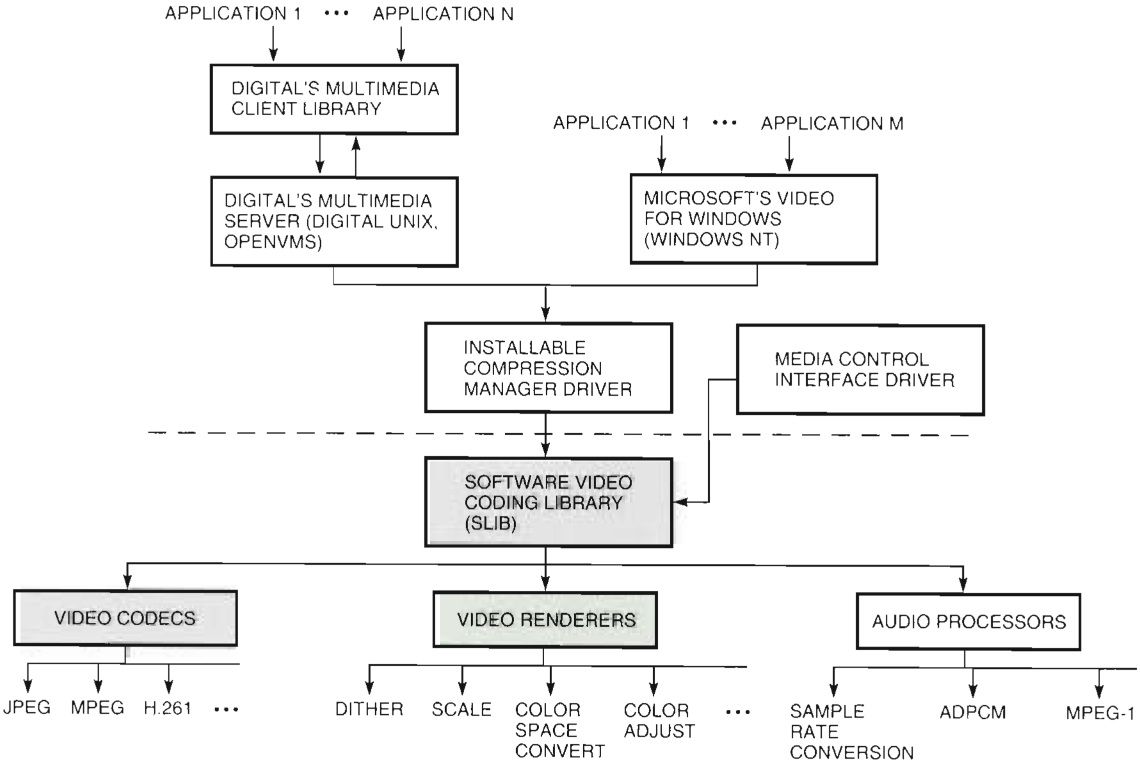

APPLICATION 1 ••• APPLICATION N
DIGITAL'S MULTIMEDIA CLIENT LIBRARY
DIGITAL'S MULTIMEDIA SERVER (DIGITAL UNIX, OPENVMS)
APPLICATION 1 ••• APPLICATION M
MICROSOFT'S VIDEO FOR WINDOWS (WINDOWS NT)
INSTALLABLE COMPRESSION MANAGER DRIVER
MEDIA CONTROL INTERFACE DRIVER
SOFTWARE VIDEO CODING LIBRARY (SLIB)
VIDEO CODECS
VIDEO RENDERERS
AUDIO PROCESSORS
JPEG
MPEG
H.261
•••
DITHER
SCALE
COLOR SPACE CONVERT
COLOR ADJUST
•••
SAMPLE RATE CONVERSION
ADPCM
MPEG-1

Figure 7
Software Video Library Hierarchy

of the codec such as whether or not it can handle a particular input format and provide information about the bit stream being processed. These routines can also be used for gathering statistics. Teardown routines, as the name suggests, are used for closing the codec and destroying all internal memory (state information) associated with it. For all video codecs, SLIB provides convenience functions to construct a table of contents containing the offsets to the start of frames in the input bit stream. These convenience functions are useful for short clips: once a table of contents is built, random access and other VCR functions can be implemented easily. (These routines are discussed further in the section on sample applications.)

Implementation of Video Codecs

In this section, we present the program flow for multimedia applications that incorporate the various video codecs. These applications are built on top of SLIB. We also discuss specific calls from the library's API to explain concepts.

Motion JPEG Motion JPEG is the de facto name of the compression scheme that uses the JPEG compression algorithm developed for still images to code video sequences. The motion JPEG (or M-JPEG) player was the first decompressor we developed. We had recently completed the Sound & Motion J300 adapter that could perform JPEG compression, decompression, and dithering in hardware.[9,10] We now wanted to develop a software decoder that would be able to decode video sequences produced by the J300 and its successor, the FullVideo Supreme JPEG adapter, which uses the peripheral component interconnect (PCI).[10] Only baseline JPEG compression and decompression have been implemented in SLIB. This is sufficient for greater than 90 percent of today's existing applications. Figure 2a and Figure 3 show the block diagrams for the baseline JPEG codec, and Figure 8 shows the flow control for compressing raw video using the video library routines. Due to the symmetric structure of the algorithm, the flow diagram for the JPEG decompressor looks very similar to the one for the JPEG compressor.

The amount of compression is controlled by the amount of quantization in the individual image frames constituting the video sequence. The coefficients for every 8-by-8 block within the image $F(x,y)$ are quantized and dequantized as

$$\begin{aligned} \mathrm{F}_q(x,y) &= \left| \frac{F(x,y)}{\mathrm{QTable}(x,y)} \right| \mathrm{F}(x,y) \\ &= \mathrm{F}_q(x,y) \times \mathrm{QTable}(x,y). \end{aligned} \tag{5}$$

In equation (5), QTable represents the quantization matrices, also called visibility matrices, associated with the frame $F(x,y)$. (Each component constituting the frame can have its own QTable.) SLIB provides routines to download QTables to the encoder explicitly; tables provided in the ISO specification can be used as defaults. The library provides a quality factor that can scale the base quantization tables, thus providing a control knob mechanism for varying the amount of compression from frame to frame. The quality factor may be dynamically varied between 0 and 10,000, with a value of 10,000 causing no quantization (all quantization table elements are equal to 1), and a value of 0 resulting in maximum quantization (all quantization table elements are equal to 255). For intermediate values:

$$\mathrm{QTable}(x,y) = \tag{6}$$
$$Clip\left(\left|\frac{VisibilityTable(x,y)\times(10^4 - QualFactor)\times 255}{10^4\times\min\{VisibilityTable(x,y)\}}\right|\right).$$

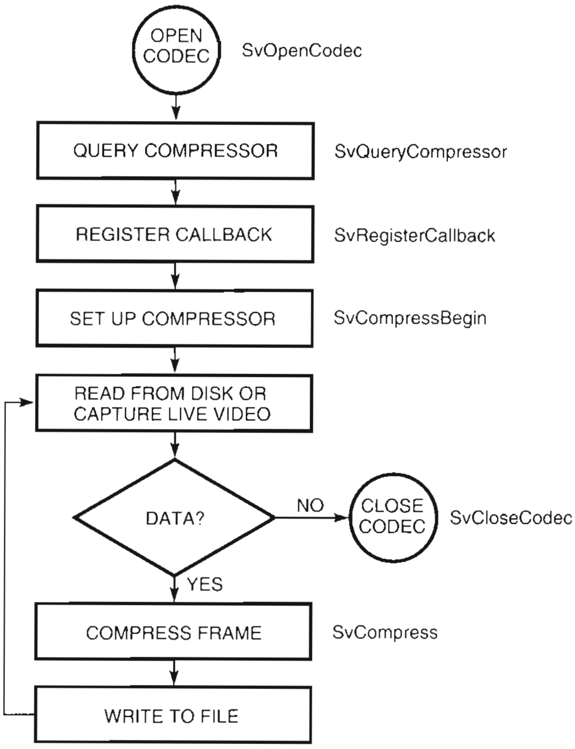


Figure 8
Flow Control for M-JPEG Compression

The Clip( ) function forces the out-of-bounds values to be either 255 or 1. At the low end of the quality setting (small values of the quality factor), the above formula produces quantization tables that cause noticeable artifacts.

Although Huffman tables do not affect the quality of the video, they do influence the achievable bit rate for a given video quality. As with quantization tables, SLIB provides routines for loading and using custom Huffman tables for compression. Huffman coding works best when the source statistics are known; in practice, statistically optimized Huffman tables are rarely used due to the computational overhead involved in their generation. In the case where these tables are not explicitly provided, the library uses as default the baseline tables suggested in the ISO specification. In the case of decompression, the tables may be present in the compressed bit stream and can be examined by invoking appropriate query calls. In the AVI format, Huffman tables are not present in the compressed bit stream, and the default ISO tables are always used.

Query routines for determining the supported input and output formats for a particular compressor are also provided. For M-JPEG compression, some of the supported input formats include interleaved 4:2:2 YUV, noninterleaved 4:2:2 YUV, interleaved and noninterleaved RGB, 32-bit RGB, and single component (monochrome). The supported output formats include JPEG-compressed YUV and JPEG-compressed single component.

ISO's MPEG-1 Video Once we had implemented the M-JPEG codec, we turned our attention to the MPEG-1 decoder. MPEG-1 is a highly asymmetric algorithm. The committee developing this standard purposely kept the decompressor simple: it was expected that there would be many cases of compress once and decompress multiple times. In general, the task of compression is much more complex than that of decompression. As of this writing, achieving real-time performance for MPEG-1 compression in software is not possible. Thus we concentrated our energies on implementing and optimizing an MPEG-1 decompressor while leaving MPEG-1 compression for batch mode. Someday we hope to achieve real-time compression all in software with the Alpha processor. Figure 9 illustrates the high-level scheme of how SLIB fits into an MPEG player. The MPEG-1 system stream is split into its audio and video substreams, and each is handled separately by the different components of the video library. Synchronization between audio and video is achieved at the application layer by using the presentation time-stamp information embedded in the system stream. A timing controller module within the application can adjust the rate at which video packets are presented to the SLIB video decoder and renderer. It can indicate to the decoder whether to skip the decoding of B- and P-frames.

Figure 10 illustrates the flow control for an MPEG-1 video player written on top of SLIB. The scheme relies on a callback function that is registered with the codec during initial setup, and a SvAddBuffers function, written by the client, which provides the codec with the bitstream data to be processed. The codec is primed by adding multiple buffers, each typically containing a single video packet from the demultiplexed system stream. These buffers are added to the codec's internal buffer queue. After enough data has been provided, the decoder is told to parse the bit stream in its buffer queue until it finds the next (first) picture. The client application can specify which type of picture to locate (I, P, or B) by setting a mask bit. After the picture is found and its information returned to the client, the client may choose to either decompress this picture or to skip it by invoking the routine to find the next picture. This provides an effective mechanism for rate control and for VCR controls such as step forward, fast forward, step back, and fast reverse. If the client requests that a non–key picture (P or B) be decompressed and the codec does not have the required reference (I or P) pictures needed to perform this operation, an error is returned. The client can then choose to abort or proceed until the codec finds a picture it can decompress.

During steady state, the codec may periodically invoke the callback function to exchange messages with the client application as it compresses or decompresses the bit stream. Most messages sent by the codec expect some action from the client. For example, one of the messages sent by the codec to the application is

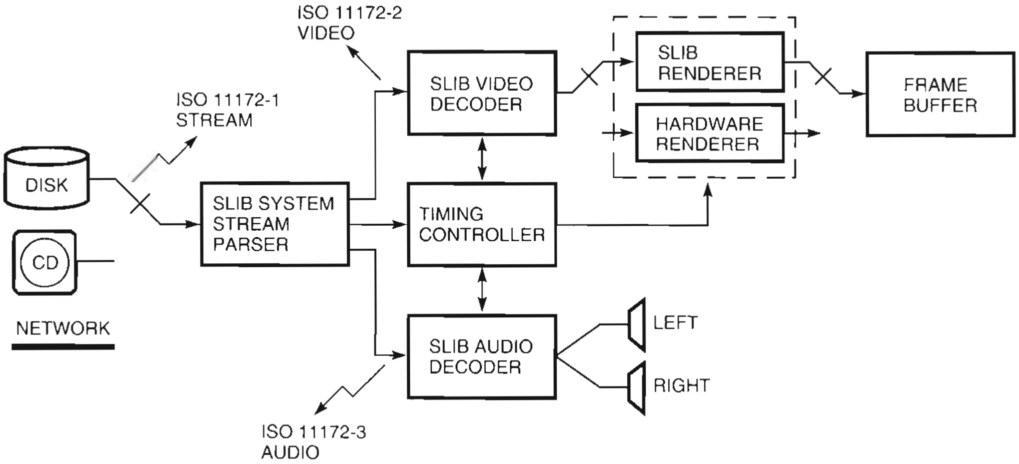


Figure 9
SLIB as Part of a Full MPEG Player

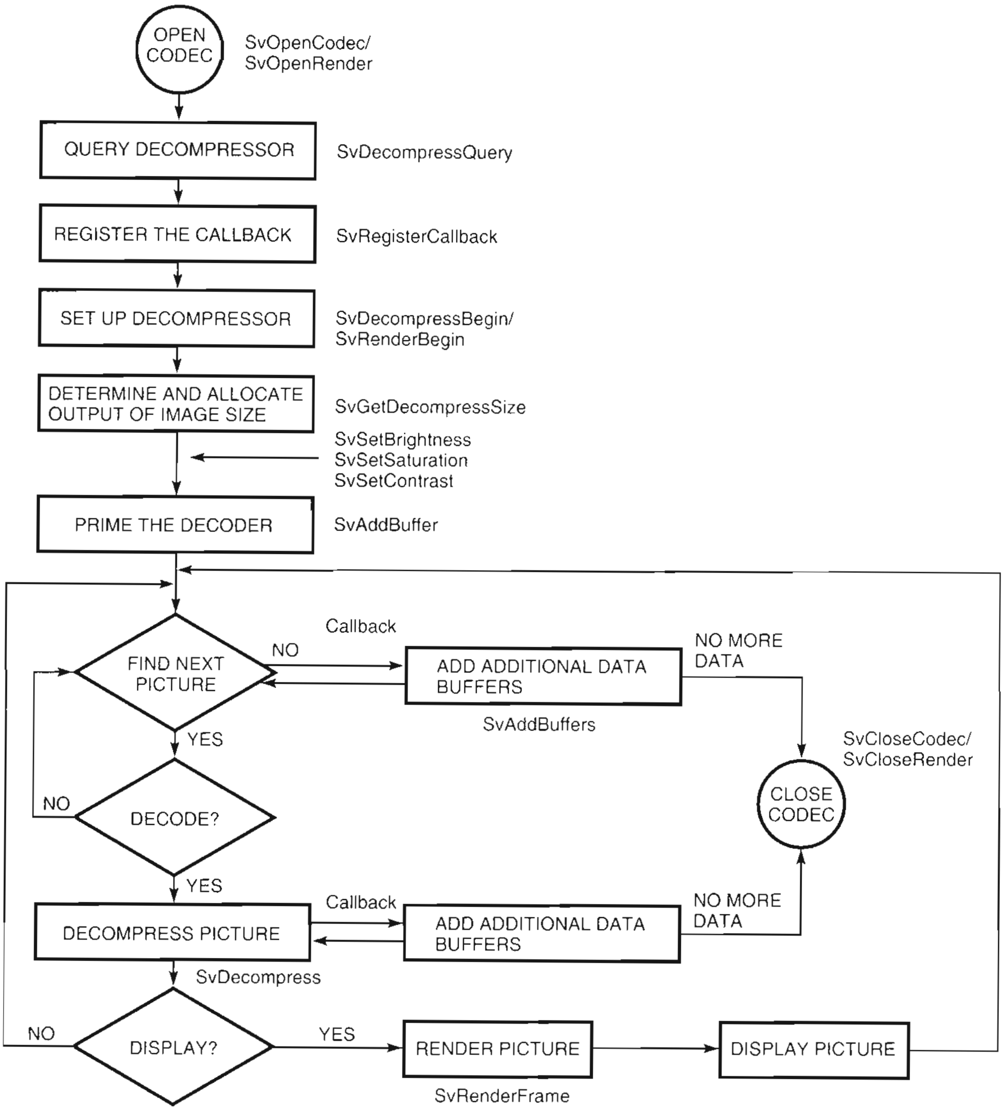


Figure 10
Flow Control for MPEG-1 Video Playback

a CB_END_BUFFERS message, which indicates the codec has run out of data and the client needs to either add more data buffers or abort the operation. Another message, CB_RELEASE_BUFFERS, indicates the codec is done processing the bit-stream data in a data buffer, and the buffer is available for client reuse. One possible action for the client is to fill this newly available buffer with more data and pass it back to the codec. In the other direction, the client may send messages to the codec through a ClientAction field. Table 1 gives some of the messages that can be sent to the codec by the application.

Another use for the callback mechanism is to accommodate client operations that need to be intermixed between video encoding/decoding operations. For example, the application may want to process audio samples while it is decompressing video. The codec can then be configured such that the callback function is invoked at a (near) periodic rate. A CB_PROCESSING message is sent to the application by the codec at regular intervals to give it an opportunity for rate control of video and/or to perform other operations.

Table 1
List of Client Messages

Message	Interpretation
CLIENT_ABORT	Abort processing of the frame
CLIENT_CONTINUE	Continue processing the frame
CLIENT_DROP	Do not decompress
CLIENT_PROCESS	Start processing

Typically the order in which coded pictures are presented to the decoder does not correspond to the order in which they are to be displayed. Consider the following example:

Display Order	I1	B2	B3	P4	B5	B6	P7	B8
Decoder Input	I1	P4	B2	B3	P7	B5	B6	I10

The order mismatch is an artifact of the compression algorithm—a B-picture cannot be decoded until both its past and future reference frames have been decoded. Similarly a P-picture cannot be decoded until its past reference frame has been decoded. To get around this problem, SLIB defines an output multibuffer. The size of this multibuffer is approximately equal to three times the size of a single uncompressed frame. For example, for a 4:2:0 subsampled CIF image, the size of the multibuffer would be 352 by 288 by 1.5 by 3 bytes (the exact size is returned by the library during initial codec setup). After steady state has been reached, each invocation to the decompress call yields the correct next frame to be displayed as shown in Figure 11. To avoid expensive copy operations, the multibuffer is allocated and owned by the software above SLIB.

ITU-T's Recommendation H.261 (a.k.a. *p* × 64) At the library level, decompressing an H.261 stream is very similar to MPEG-1 decoding with one exception: instead of three types of pictures, the H.261 recommendation defines only two, key frames and non–key frames (no bidirectional prediction). The implication for implementation is that the size of the multibuffer is approximately twice the size of a single decompressed frame. Furthermore, the order in which compressed frames are presented to the decompressor is the same as the order in which they are to be displayed.

To satisfy the H.261 recommendation, SLIB implements a streaming interface for compression and decompression. In this model, the application feeds input buffers to the codec, which processes the data in the buffers and returns the processed data to the application through a callback routine. During decompression, the application layer passes input buffers containing sections of an H.261 bit stream. The bit stream can be divided arbitrarily, or, in the case of live teleconferencing, each buffer can contain data from a transmission packet. Empty output buffers are also passed to the codec to fill with reconstructed images. Picture frames do not have to be aligned on buffer boundaries. The codec parses the bit stream and, when enough data is available, reconstructs an image. Input buffers are freed by calling the callback routine. When an image is reconstructed, it is placed in an output buffer and the buffer is returned to the application through the callback routine. The compression process is similar, but input buffers contain images and output buffers contain bit-stream data. One advantage to this streaming interface is that the application layer does not need to know the syntax of the H.261 bit stream. The codec is responsible for all bit-stream parsing. Another advantage is that the callback mechanism for returning completed images or bit-stream buffers allows the application to do other tasks without implementing multithreading.

SLIB's architecture and API can easily accommodate ISO's MPEG-2 and ITU-T's H.263 video compression algorithms because of their similarity to the MPEG-1 and H.261 algorithms.

Implementation of Video Rendering

Our software implementation of video rendering essentially parallels the hardware realization detailed elsewhere in this issue.[9] As with the hardware implementation, the software renderer is fast and simple because the complicated computations are performed off line in building the various look-up tables. In both hardware and software cases, a shortcut is achieved by dithering in YUV space and then converting to some small number of RGB index values in a look-up table.[16]

Although in most cases the mapping values in the look-up tables remain fixed for the duration of the run, the video library provides routines to dynamically adjust image brightness, contrast, saturation, and the number of colors. Image scaling is possible but affects performance. When quality is important, the software performs scaling before dithering and when speed is the primary concern, it is done after dithering.

Optimizations

We approached the problem of optimization from two directions: Platform-independent optimizations, or algorithmic enhancements, were done by exploiting knowledge of the compression algorithm and the

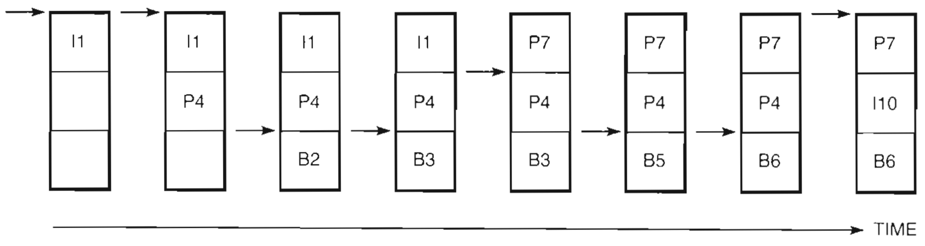


Figure 11
Multibuffering in SLIB

input data stream. Platform-dependent optimizations were done by examining the services available from the underlying operating system and by evaluating the attributes of the system's processor.

As can be seen from Table 2, the DCT is one of the most computationally intensive components in the compression pipeline. It is also common to all five international standards. Therefore, a special effort was made in choosing and optimizing the DCT. Since all five standards call for the inverse DCT (IDCT) to be postprocessed with inverse quantization, significant algorithmic savings were obtained by computing a scalar multiple of the DCT and merging the appropriate scaling into the quantizer. The DCT implemented in the library is a modified version of the one-dimensional scaled DCT proposed by Arari et al.[18] The two-dimensional DCT is obtained by performing a one-dimensional DCT on the columns followed by a one-dimensional DCT on the rows. A total of 80 multiplies and 464 adds are needed for a fully populated 8-by-8 block. In highly compressed video, the coefficient matrix to be transformed is generally sparse because a large number of elements are "zeroed" out due to heavy quantization. We exploit this fact to speed up the DCT computations. In the decoding process, the Huffman decoder computes and passes to the IDCT a list of rows and columns that are all zeros. The IDCT then simply skips these columns.[19] Another optimization uses a different IDCT, depending on the number of nonzero coefficients. The overall speedup due to these techniques is dependent on the amount of compression. For lightly compressed video, we observed that the overhead due to these techniques slowed down the decompressor. We overcame this difficulty by building into SLIB the adaptive selection of the appropriate optimization based on continuous statistics gathering. Run-time statistics of the number of blocks per frame that are all zeros are maintained, and the number of frames over which these statistics are evaluated is provided as a parameter for the client applications. Statistic gathering is minimal: a counter update and an occasional compare.

The second component of the video decoders we looked at was the Huffman decoder. Analysis of the compressed data indicated that short-code-length symbols were a large part of the compressed bit stream. The decoder was written to handle frequently occurring very short codes (< 4 bits) as special cases, thus avoiding loads from memory. For short codes (< 8 bits), look-up tables were used to avoid bit-by-bit decoding. Together, these two classes of codes account for well over 90 percent of the total collection of the variable-length codes.

A third compute-intensive operation is raster-to-block conversion in preparation for compression. This operation had the potential of slowing down the compressor on Alpha-based systems on which byte and short accesses are done indirectly. We implemented an assembly language routine that would read the uncompressed input color image and convert it to three one-dimensional arrays containing 8-by-8 blocks in sequence. Special care was taken to keep memory references aligned. Relevant bytes were obtained through shifting and masking operations. Level shifting was also incorporated within the routine to avoid touching the same data again.

Other enhancements included replacing multiplies and divides with shifts and adds, avoiding integer to floating-point conversions, and using floating-point operations wherever possible. This optimization is particularly suited to the Alpha architecture, where floating-point operations are significantly faster than integer operations. We also worked to reduce memory bandwidth. Ill-placed memory accesses can stall the processor and slow down the computations. Instructions generated by the compiler were analyzed and sometimes rescheduled to void data hazards, to keep the on-chip pipeline full, and to avoid unnecessary loads and stores. Critical and small loops were unrolled to make better use of floating-point pipelines. Reordering the computations to reuse data already in registers and caches helped minimize thrashing in the cache and the translation lookaside buffer. Memory was accessed through offsets rather than pointer

Table 2
Typical Contributions of the Major Components in the Playback of Compressed Video (SIF)

Coding Scheme	Bit-stream Parser	Huffman and Run-length Decoder	Inverse Quantizer	IDCT	Motion Compression, Block to Raster	Vector Quantization (INDEO only)	Tone Adjust, Dither, Quantize and Color Space Convert	Display
M-JPEG decode	0.8%	12.4%	10.5%	35.2%	—	—	33.7%	7.4%
MPEG-1 decode	0.9%	13.0%	9.7%	19.7%	20.2%	—	31.4%	5.1%
INDEO decode	1.0%	—	—	—	—	57.5%	36.0%	5.5%

increments. More local variables than global variables were used. Wherever possible, fixed values were hard coded instead of using variables that would need to be computed. References were made to be 32-bit or 64-bit aligned accesses instead of byte or short.

Consistent with one of the design goals, SLIB was made thread-safe and fully reentrant. The Digital UNIX, the OpenVMS, and Microsoft's Windows NT operating systems all offer support for multithreaded applications. Applications such as video playback can improve their performance by having separate threads for reading, decompressing, rendering, and displaying. Also, a multithreaded application scales up well on a multiprocessor system. Global multithreading is possible if the library code is reentrant or thread-safe. When we were trying to multithread the library internals, we found that the overhead caused by the birth and death of threads, the increase in memory accesses, and the fragmentation of the codec pipeline caused operations to slow down. For these reasons, routines within SLIB were kept single-threaded. Other operating-system optimizations such as memory locking, priority scheduling, nonpreemption, and faster timers that are generally good for real-time applications were experimented with but not included in our present implementation.

Performance on Digital's Alpha Machines

Measuring the performance of video codecs is generally a difficult problem. In addition to the usual dependencies such as system load, efficiency of the underlying operating system, and application overhead, the speed of the video codecs is dependent on the content of the video sequence being processed. Rapid movement and action scenes can delay both compression and decompression, while slow motion and high-frequency content in a video sequence can generally result in faster decompression. When comparing the performance of one codec against another, the analyst must make certain that all codecs process the same set of video sequences under similar operating conditions. Since no sequences have been accepted as standard, the analyst must decide which sequences are most typical. Choosing a sequence that favors the decompression process and presenting those results is not uncommon, but it can lead to false expectations. Sequences with similar peak signal-to-noise ratio (PSNR) may not be good enough, because more often than not PSNR (or equivalently the mean square error) does not accurately measure signal quality. With these thoughts in mind, we chose some sequences that we thought were typical and used these to measure the performance of our software codecs. We do not present comparative results to codecs implemented elsewhere since we did not have access to these codecs and hence could not test these with the same sequences.

Table 3 presents the characteristics of the three video sequences used in our experiments. Let $L_i(x,y)$ and $\hat{L}_i(x,y)$ represent the luminance component of the original and the reconstructed frame i; let n and m represent the horizontal and vertical dimensions of a frame; and let N be the number of frames in the video sequence. Then the Compression Ratio, the average output BitsPerPixel, and the average PSNR are calculated as

$$\text{Compression Ratio} = \frac{\sum_{i=1}^{N} \text{bits in frame}[i] \text{ of original video}}{\sum_{i=1}^{N} \text{bits in frame}[i] \text{ of compressed video}} \qquad (7)$$

$$\text{Avg. BitsPerPixel} = \frac{1}{N \times n \times m} \sum_{i=1}^{N} \text{bits in frame}[i] \text{ of compressed video} \qquad (8)$$

$$\text{Avg. PSNR} = 20\log_{10} \frac{255}{\frac{1}{N}\sum_{i=1}^{N} \left(\sqrt{\frac{1}{nm}\sum_{x=1}^{n}\sum_{y=1}^{m} \left[L_i(x,y) - \hat{L}_i(x,y) \right]^2} \right)}. \qquad (9)$$

Figure 12 shows the PSNR for individual frames in the video sequences along with the distribution of frame size for each of three test sequences. Frame dimensions within a sequence always remain constant.

Table 4 provides specifications of the workstations and PCs used in our experiments for generating the various performance numbers. The 21064 chip is Digital's first commercially available Alpha processor. It has a load-store architecture, is based on a 0.75-micrometer complementary metal-oxide semiconductor (CMOS) technology, contains 1.68 million transistors, has a 7- and 10-stage integer and floating-point pipeline, has separate 8-kilobyte instruction and data caches, and is designed for dual issue. The 21064A microprocessor has the same architecture as the 21064 but is based on a 0.5-micrometer CMOS technology and supports faster clock rates.

We provide performance numbers for the video sequences characterized in Table 3. Figure 13 provides measured data on CPU usage when compressed video (from Table 3) is played back at 30 frames per second on the various test platforms shown in Table 4. We chose "percentage of CPU used" as a measure of performance because we wanted to know whether the CPU could handle any other tasks when it was doing video processing. Fortunately, it turned out the

Table 3
Characteristics of the Video Sequences Used to Generate the Performance Numbers Shown in Figure 12

Name	Compression Algorithm	Spatial Resolution (width X height)	Temporal Resolution (No. of Frames)	Avg. BitsPerPixel	Compression Ratio	Avg. PSNR (dB)
Sequence 1	M-JPEG	352 × 240	200	0.32	50:1	31.56
Sequence 2	MPEG-1 Video	352 × 288	200	0.17	69:1	32.28
	M-JPEG	352 × 240	200	0.56	28:1	31.56
Sequence 3	INDEO	352 × 240	200	0.16	47:1	28.73

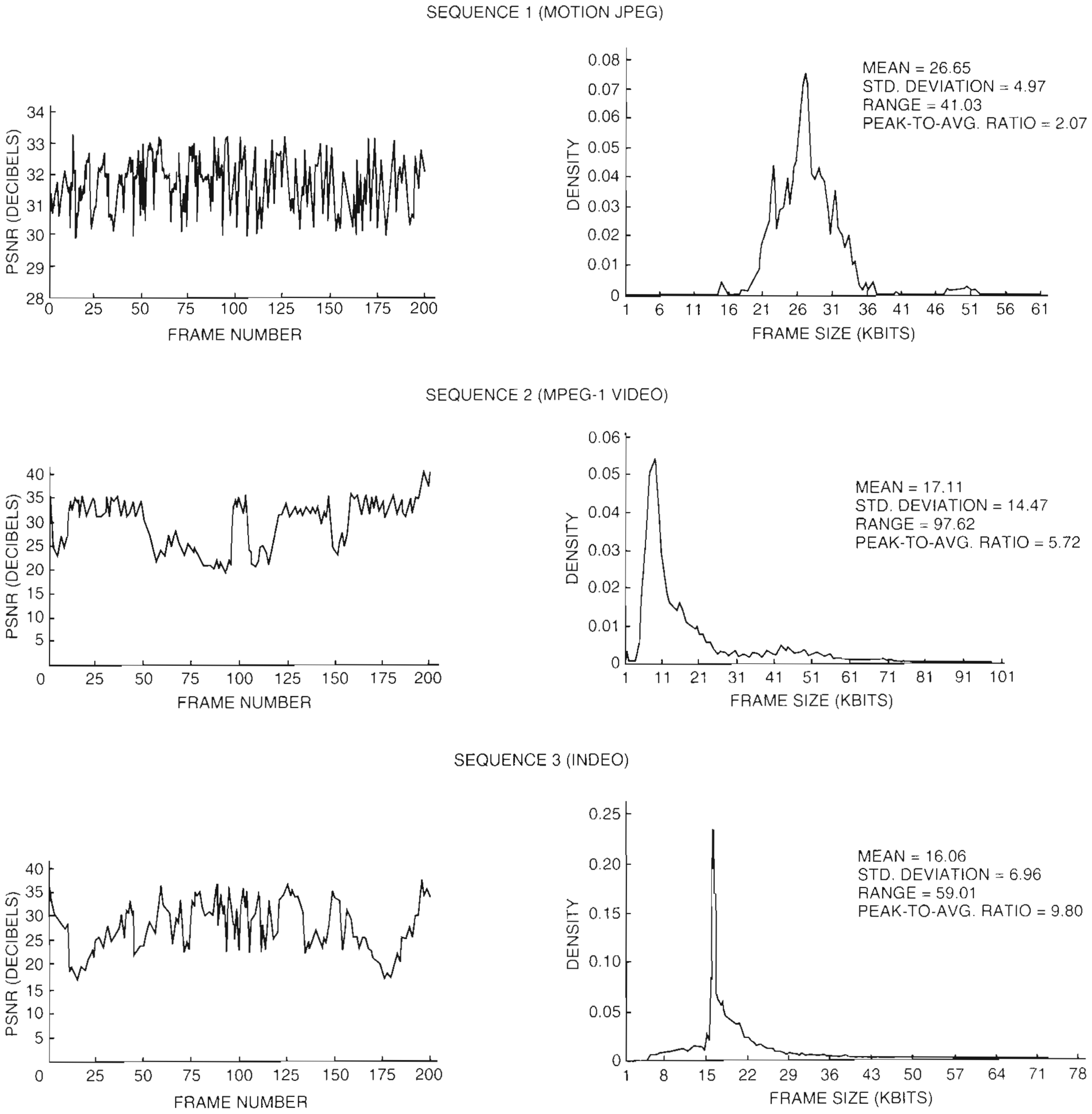


Figure 12
Characteristics of the Three Test Sequences

Table 4
Specifications of Systems Used in Experimentation

System Name	CPU	Bus	Clock Rate	Cache	Memory	Operating System	Disk
AlphaStation 600 5/266 workstation	Alpha 21164A	PCI	266 MHz (3.7 ns)	2 MB	64 MB	Digital UNIX V3.2	RZ28B
AlphaStation 200 4/266 workstation	Alpha 21064A	PCI	266 MHz (3.7 ns)	2 MB	32 MB	Digital UNIX V3.0	RZ58
DEC 3000/M900 workstation	Alpha 21064A	TURBOchannel	275 MHz (3.6 ns)	2 MB	64 MB	Digital UNIX V3.2	RZ58
DEC 3000/M500 workstation	Alpha 21064	TURBOchannel	133 MHz (7.5 ns)	512 KB	32 MB	Digital UNIX V3.0	RZ57

answer was a resounding "Yes" in the case of Alpha processors. The video playback rate was measured with software video rendering enabled. When hardware rendering is available, estimated values for video playback are provided.

From Figure 13, it is clear that today's workstations are capable of playing SIF video at full frame rates with no hardware acceleration. High-quality M-JPEG and MPEG-1 compressed video clips can be played at full speed with 20 percent to 60 percent of the CPU available for other tasks. INDEO decompression is faster than M-JPEG and MPEG due to the absence of DCT processing. (INDEO uses a vector quantization method based on pixel differencing.) On three out of

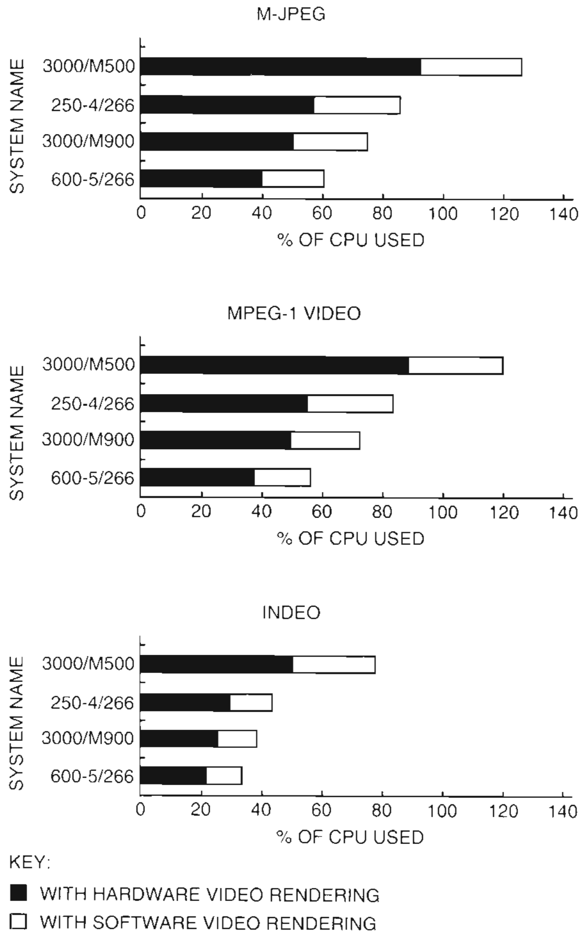


Figure 13
Percentage of CPU Required for Real-time Playback at 30 fps on Four Different Alpha based Systems

the four machines tested, two SIF INDEO clips could be played back at full speed with CPU capacity left over for other tasks.

The data also shows the advantage of placing the color conversion and rendering of the video in the graphics hardware (see Table 2 and Figure 13). Software rendering accounts for one-third of the total playback time. Since rendering is essentially a table look-up function, it is a good candidate for moving into hardware. If hardware video rendering is available, multiple M-JPEG and MPEG-1 clips can be played back on three of the four machines on which the software was tested.

Software video compression is more time-consuming than decompression. All algorithms discussed in this paper are asymmetric in the amount of processing needed for compression and decompression. Even though the JPEG algorithm is theoretically symmetric, the performance of the JPEG decoder is better than that of the encoder. The difference in performance is due to the sparse nature of the quantized coefficient matrices, which is exploited by the appropriate IDCT optimizations.

For video encoders, we measured the rate of compression for both SIF and quarter SIF (QSIF) formats. Since the overhead due to I/O affects the rate at which the compressor works, we present measured rates collected when the raw video sequence is read from disk and when it is captured in real time. The capture cards used in our experiments were the Sound & Motion J300 (for systems with the TURBOchannel bus) and the FullVideo Supreme (for PCI-based systems). The compressed bit streams were stored as AVI files on local disks. The sequences used in this experiment were the same ones used for obtaining measurement for the various decompressors; their output characteristics are given in Table 3. Table 5 provides performance numbers for the M-JPEG and an unoptimized INDEO compressor. For M-JPEG, rates for both monochrome and color video sequences are provided.

The data in Table 5 indicates that the M-JPEG compression outperforms INDEO (although one has to keep in mind that INDEO was not optimized). This difference occurs because M-JPEG compression, unlike INDEO, does not rely on interframe prediction or motion estimation for compression. Furthermore, when raw video is compressed from disk, the encoder performs better than when it is captured and compressed in real time. This can be explained on the basis of the overhead resulting from context switching in the operating system and the scheduling of sequential capture operation by the applications. Real-time capture and compression of image sizes larger than QSIF still require hardware assistance. It should be noted that in Table 5, the maximum compression rate for real-time capture and compression does not exceed 30 frames per second, which is the limit of the capture hardware. Since there are no such limitations for disk reads, compression rates of greater than 30 frames per second for QSIF sequences are recorded.

With the newer Alpha chip we expect to see improved performance. A factor we neglected in our calculations was prefiltering. Some capture boards are capable of capturing only in CCIR 601 format and do not include decimation filters as part of their hardware. In such cases, the software has to filter each frame down to CIF or QCIF, which adds substantially to the overall compression time. For applications that do not require real-time compression, software digital-video compression may be a viable solution since video can be captured on fast disk arrays and compressed later.

Table 5
Typical Number of Frames Compressed per Second

	M-JPEG (Color)				M-JPEG (Monochrome)				INDEO (Color)			
System	Compress (fps)		Capture and Compress (fps)		Compress (fps)		Capture and Compress (fps)		Compress (fps)		Capture and Compress (fps)	
	SIF	QSIF	SIF	QSIF	SIF	QSIF	SIF	QSIF	SIF	QSIF	SIF	QSIF
AlphaStation 600 5/266 workstation	21.0	79.4	20.0	30.0	32.8	130	29.0	30.0	8.7	35.4	5.8	23.0
AlphaStation 200 4/266 workstation	10.8	45.1	12.0	30.0	15.8	72.9	20.0	30.0	5.6	22.0	4.2	13.0
DEC 3000/M900 workstation	13.2	56.6	7.9	28.0	21.9	87.8	14.0	29.0	6.0	25.4	4.5	7.6
DEC 3000/M500 workstation	6.7	26.6	7.3	8.1	10.4	40.4	7.4	8.2	2.8	11.8	2.2	8.7

Sample Applications

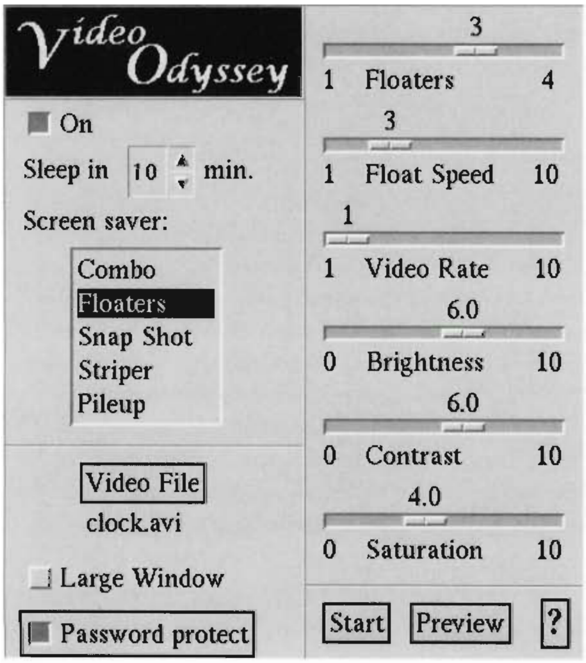
We implemented several applications to test our architecture (codecs and renderer) and to create a test bed for performance measurements. These programs also served as sample code for software developers incorporating SLIB into other multimedia software layers.

The Video Odyssey Screen Saver

The Video Odyssey screen saver uses software video decompression and 24-bit YCbCr to 8-bit pseudocolor rendering to deliver video images to the screen in a variety of modes. The program is controlled by a control panel, shown in Figure 14.

The user can select from several methods of displaying the decompressed video or let the computer cycle through all methods. The floaters mode, shown in Figure 15, floats one to four copies of the video around the screen with the number of floating windows controlled by a slider in the control panel. The snapshot mode floats one window of the video around the screen, but every second takes a snapshot of a frame and pastes it to the background behind the floating window.

All settings in the control panel are saved in a configuration file in the user's home directory. The user selects a video file with the file button. In the current implementation, any AVI file containing Motion JPEG or raw YUV video is acceptable. The user can set the time interval for the screen saver to take over. Controls for setting brightness, contrast, and saturation are also provided. Video can be played back at normal resolution or with ×2 scaling. Scaling is integrated with the color conversion and dithering for optimization. A pause feature allows the user to leave his or her screen in a locked state with an active screen saver. The screen is unlocked only if the correct password is provided.



■ On

Sleep in 10 min.

Screen saver:

- Combo
- Floaters
- Snap Shot
- Striper
- Pileup

Video File

clock.avi

□ Large Window

■ Password protect

3
1 Floaters 4

3
1 Float Speed 10

1
1 Video Rate 10

6.0
0 Brightness 10

6.0
0 Contrast 10

4.0
0 Saturation 10

Figure 14
Video Odyssey Control Panel

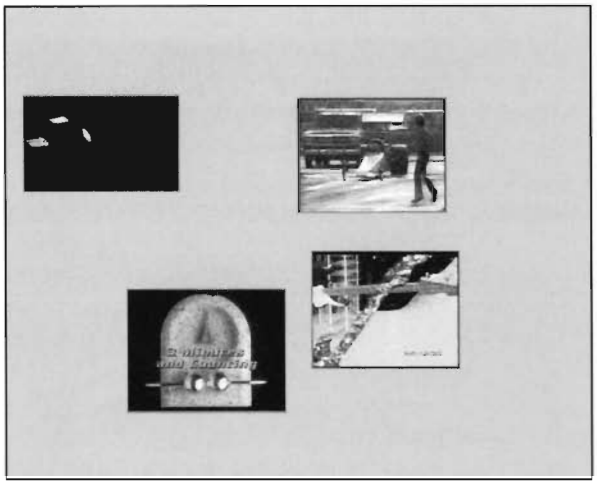
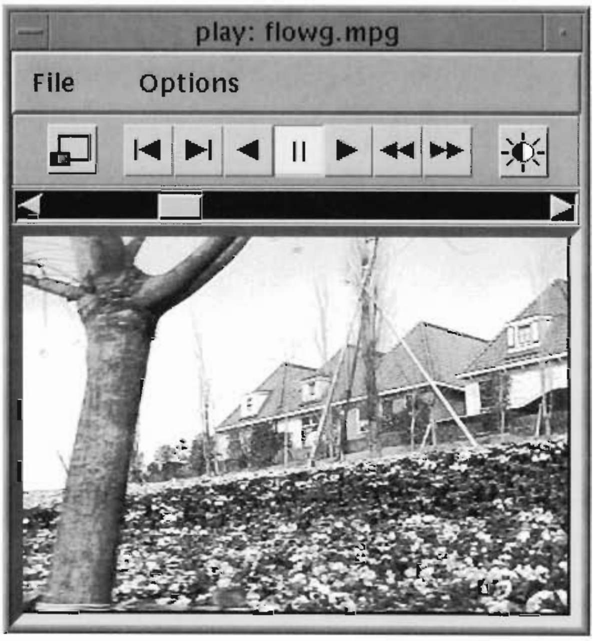


Figure 15
Video Odyssey Screen Saver in Floaters Mode

The Software Video Player

The software video player is an application for viewing video that is similar to a VCR. Like Video Odyssey, the software video player exercises the decompression and rendering portions of SLIB. Unlike Video Odyssey, the software video player allows random access to any portion of the video and permits single-step, reverse, and fast-forward functions. Figure 16 shows the display window of the software video player.



play: flowg.mpg

Figure 16
The Software Video Player Display Window

The user moves through the file with a scroll bar and a set of VCR-like buttons. The button on the far left of the display window allows the video to be displayed at normal size or at a magnification of ×2. The far-right button allows adjustment of brightness, contrast, saturation, and number of displayed colors. The quality of the dithering algorithm used in rendering is such that values as low as 25 colors lead to acceptable image quality. Allowable file formats for the software video player are M-JPEG (AVI format and the JPEG file interchange format or JFIF), MPEG-1 (both video and system streams), and raw YUV.

Random access into the file is done in one of two ways, depending on the file format. For formats that contain an index of the frame positions in the file (like AVI files), the index is simply used to seek the desired frame. For formats that do not contain an index, such as MPEG-1 and JFIF, the software video player estimates the location of a frame based on the total length of the video clip and a running average of frame size. This technique is adequate for most video clips and has the advantage of avoiding the time needed to first build an index by scanning through the file.

Interframe compression schemes like MPEG-1 and INDEO pose special problems when trying to access a random frame in a video clip. MPEG-1's B- and P-frames are dependent on preceding frames and cannot be decompressed alone. One technique for handling random access into files with non–key frames and no frame index is to use the file position specified by the user (with a scroll bar or by other means) as a starting point and then to search the bit stream for the next key frame (an I-frame in MPEG-1). At that point, display can proceed normally. Reverse play is also a problem with these formats. The software video player deals with reverse by displaying only the key frames. It could display all frames in reverse by predecompressing all frames in a group and then displaying them in reverse order, but this would require large amounts of memory and would pose problems with processing delays. Rate control functions, including fast-forward and fast-reverse functions, can be done by selectively throwing out non–key frames and processing key or I-frames only.

Other Applications

Several other applications using different components of SLIB were also written. Some of these are (1) Encode—a video encoding application that uses SLIB's compression component to compress raw video to M-JPEG format, (2) Rendit—a viewer for true color images that uses SLIB's rendering component to scale, tone-adjust, dither, quantize, color space convert, and display 24-bit RGB or 16-bit YUV images on frame buffers with limited planes, and (3) routines for viewing compressed on-line video documentation that was incorporated into Digital's videoconferencing product.

Related Work

While considerable effort has been devoted to optimizing video decoders, little has been done for video encoders. Encoding is generally computationally more complex and time-consuming than decoding. As a result, obtaining real-time performance from encoders has not been feasible. Another rationalization for interest in decoders has been that many applications require video playback and only a few are based on video encoding. As a result, "code once, play many times" has been the dominant philosophy. In most papers, researchers have focused on techniques for optimizing the various codecs; very little has been published on providing a uniform architecture and an intuitive API for the video codecs.

In this section, we present results from other papers published on software video codecs. Of the three international standards, MPEG-1 has attracted the most attention, and our presentation is biased slightly toward this standard. We concentrate on work that implements at least one of the three recognized international standards.

The JPEG software was made popular by the Independent Software JPEG Group formed by Tom Lane.[20] He and his colleagues implemented and made available free software that could perform baseline JPEG compression and decompression. Considerable attention was given to software modularity and portability. The main objective of this codec was still-image compression although its modified version has been used for decompression of motion JPEG sequences as well.

The MPEG software video decoder was made popular by the multimedia research group at the University of California, Berkeley. The availability of this free software sparked the interest of many who now had the opportunity to play with and experiment with compressed video. Patel et al. describe the implementation of this software MPEG decoder.[21] The focus in their paper is on an MPEG-1 video player that would be portable and fast. The authors describe various optimizations, including in-line procedures, custom coding frequent bit-twiddling operations, and rendering in the YUV space with color conversion through look-up tables. They observed that the key bottleneck toward real-time performance was not the computation involved but the memory bandwidth. They also concluded that data structure organization and bit-level manipulations were critical for good performance. The authors propose a novel metric for comparing the performance of the decoder on systems marketed by different systems vendors. Their metric, the percentage of required bit rate per second per thousand dollars (PBSD), takes into account the price of the system on which the decoder is being evaluated.

Bheda and Srinivasan describe the implementation of an MPEG-1 decoder that is portable across platforms because the software is written entirely in a high-level language.[22] The paper describes the various optimizations done to improve the decoder's speed and provides performance numbers in terms of number of frames displayed per second. The authors compare the speed of their decoder on various platforms, including Digital's first Alpha-based PC running Microsoft's Windows NT system. They conclude that their decoder performed best on the Alpha system. It was able to decompress, dither, and display a 320-pixel by 240-line video sequence at a rate of 12.5 frames per second. A very brief description of the API supported by the decoder is also provided. The API is able to support operations such as random access, fast forward, and fast reverse. Optional skipping of B-frames is possible for rate control. The authors conclude that the size of the cache and the performance of the display subsystem are critical for real-time performance.

Bhaskaran and Konstantinides describe a real-time MPEG-1 software decoder that can play both audio and video data on a Hewlett-Packard PA-RISC processor-based workstation.[23] The paper provides step-by-step details on how optimization was carried out at both the algorithmic and the architectural levels. The basic processor was enhanced by including in the instruction set several multimedia instructions capable of performing parallel arithmetic operations that are critical in video codecs. The display subsystem is able to handle color conversion of YCbCr data and up-sampling of image data. The performance of the decoder is compared to software decoders running on different platforms from different manufacturers. The comparison is not truly fair because the authors compare their decoder, which has hardware assistance available to it (i.e., an enhanced graphic subsystem and new processor instructions), to other decoders that are truly software based. Furthermore, since all the codecs were not running on the same machine under similar operating conditions and since the sequence tested on their decoder is not the same as the one used by the others, the comparison is not truly accurate. The paper does not provide any information on the programming interface, the control flow, and the overall software architecture.

There are numerous other descriptions of the MPEG-1 software codecs. Eckart describes a software MPEG video player that is capable of decoding both audio and video in real time on a PC with a 90-megahertz Pentium processor.[24] Software for this decoder is available freely over the Internet. Gong and Rowe describe a parallel implementation of the MPEG-1 encoder that runs on a network of workstations.[25] The performance improvements of greater than 650 percent are reported when the encoding process is performed on 9 networked HP 9000/720 systems as compared to a single system.

Wu et al. describe the implementation and performance of a software-only H.261 video codec on the PowerPC 601 reduced instruction set computer (RISC) processor.[26] This paper is interesting in that it deals with optimizing both the encoder and the decoder to facilitate real-time, full-duplex network connections. The codec plugs under the QuickTime architecture developed by Apple Computer, Inc. and can be invoked by applications that have programmed to the QuickTime interface. The highest display rate is slightly under 18 frames per second for a QSIF video sequence coded at 64 kilobits per second with disk access. With real-time video capture included, the frame rate reduces to between 5 and 10 frames per second. The paper provides an interesting insight by giving a breakdown of the amount of time spent in each stage of coding and decoding on a complex instruction set computer (CISC) versus a RISC system. Although the paper does a good job of describing the optimizations, very little is mentioned about the software architecture, the programming interface, and the control flow.

We end this section by recommending some sources for obtaining additional information on the state of the art in software-only video in particular and in multimedia in general. First, the Society of Photo-Optical Instrumentation Engineers (SPIE) and the Association of Computing Machinery (ACM) sponsor annual multimedia conferences. The proceedings from these conferences provide a comprehensive record of the advances made on a year-to-year basis. In addition, both the Institute of Electrical and Electronics Engineers (IEEE) and ACM regularly publish issues devoted to multimedia. These special issues contain review papers with sufficient technical details.[14,27] Finally, an excellent book on the subject of video compression is the recently published *Digital Pictures* (second edition) by Arun Netravali and Barry Haskel from Plenum Press.

Conclusions

We have shown how popular video compression schemes are composed of an interconnection of distinct functional blocks put together to meet specified design objectives. The objectives are almost always set by the target applications. We have demonstrated that the video rendering subsystem is an important component of a complete playback solution and presented a novel algorithm for mapping out-of-range colors.

We described the design of our software architecture for video compression, decompression, and playback. This architecture has been successfully implemented over multiple platforms, including the Digital UNIX, the OpenVMS, and Microsoft's Windows NT operating systems. Performance results corroborate our claim that current processors can adequately handle playback of compressed video in real time with little or no hardware assistance. Video compression, on the other hand, still requires some hardware assistance for real-time performance. We believe the widespread use of video on the desktop is possible if high-quality video can be delivered economically. By providing software-only video playback, we have taken a step in this direction.

References

1. A. Akansu and R. Haddad, "Signal Decomposition Techniques: Transforms, Subbands, and Wavelets," *Optcon '92, Short Course 28* (November 1992).

2. E. Feig and S. Winograd, "Fast Algorithms for Discrete Cosine Transform," *IEEE Transactions on Signal Processing*, vol. 40, no. 9 (1992): 2174–2193.

3. S. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28 (1982): 129–137.

4. J. Max, "Quantizing for Minimum Distortion," *IRE Transactions on Information Theory*, vol. 6, no. 1 (1960): 7–12.

5. N. Nasrabadi and R. King, "Image Coding using Vector Quantization: A Review," *IEEE Transactions Communications*, vol. 36, no. 8 (1988): 957–971.

6. D. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE*, vol. 40 (1952): 1098–1101.

7. H. Harashima, K. Aizawa, and T. Saito, "Model-based Analysis-Synthesis Coding of Video Telephone Images—Conception and Basic Study of Intelligent Image Coding," *Transactions of IEICE*, vol. E72, no. 5 (1981): 452–458.

8. *Information Technology—Digital Compression and Coding of Continuous-tone Still Images, Part 1: Requirements and Guidelines*, ISO/IEC IS 10918-1:1994 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1994).

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. P. Bahl, "The J300 Family of Video and Audio Adapters: Software Architecture," *Digital Technical Journal*, vol. 7, no. 4 (1995, this issue): 34–51.

11. *Video Codec for Audiovisual Services at p × 64 kbits/s*, ITU-T Recommendation H.261, CDM XV-R 37-E (Geneva: International Telegraph and Telephone Consultative Committee, 1990).

12. *Coding of Moving Pictures and Associated Audio for Digital Storage Media at Up to about 1.5 Mbits/s*, ISO/IEC Standard 11172-2:1993 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1993).

13. *Generic Coding of Moving Pictures and Associated Audio, Recommendation H.262*, ISO/IEC CD 13818-2:1994 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1994).

14. Special Issue on Advances in Image and Video Compression, *Proceedings of the IEEE* (February 1995).

15. R. Ulichney, "Video Rendering," *Digital Technical Journal*, vol. 5, no. 2 (Spring 1993): 9–18.

16. L. Seiler and R. Ulichney, "Integrating Video Rendering into Graphics Accelerator Chips," *Digital Technical Journal*, vol. 7, no. 4 (1995, this issue): 76–88.

17. R. Ulichney, "Method and Apparatus for Mapping a Digital Color Image from a First Color Space to a Second Color Space," U.S. Patent 5,233,684 (1993).

18. Y. Arari, T. Agui, and M. Nakajima, "A Fast DCT-SQ Scheme for Images," *IEEE Transactions IEICE*, E-71 (1988): 1095–1097.

19. K. Froitzheim and K. Wolf, "Knowledge-based Approach to JPEG Acceleration," Digital Video Compression: Algorithms and Technologies 1995, *Proceedings of the SPIE*, vol. 2419 (1995): 2–13.

20. T. Lane, "JPEG Software," Independent JPEG Group, unpublished paper available on the Internet.

21. K. Patel, B. Smith, and L. Rowe, "Performance of a Software MPEG Video Decoder," *Proceedings of ACM Multimedia '93*, Anaheim, Calif. (1993): 75–82.

22. H. Bheda and P. Srinivasan, "A High-performance Cross-platform MPEG Decoder," Digital Video Compression: Algorithms and Technologies 1995, *Proceedings of the SPIE*, vol. 2187 (1994): 241–248.

23. K. Bhaskaran and K. Konstantinides, "Real-Time MPEG-1 Software Decoding on HP Workstations," Digital Video Compression: Algorithms and Technologies 1995, *Proceedings of the SPIE*, vol. 2419 (1995): 466–473.

24. S. Eckart, "High Performance Software MPEG Video Playback for PCs," Digital Video Compression: Algorithms and Technologies 1995, *Proceedings of the SPIE*, vol. 2419 (1995): 446–454.

25. K. Gong and L. Rowe, "Parallel MPEG-1 Video Encoding," *Proceedings of the 1994 Picture Coding Symposium*, Sacramento, Calif. (1994).

26. H. Wu, K. Wang, J. Normile, D. Ponceleon, K. Chu, and K. Sung, "Performance of Real-time Software-only H.261 Codec on the Power Macintosh," Digital Video Compression: Algorithms and Technologies 1995, *Proceedings of the SPIE*, vol. 2419 (1995): 492–498.

27. Special Issue on Digital Multimedia Systems, *Communications of the ACM*, vol. 34, no. 1 (April 1991).

Biographies

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.

Paul S. Gauthier
Paul Gauthier is the president and founder of Image Softworks, a software development company in Westford, Massachusetts, specializing in image and video processing. He received a Ph.D. in physics from Syracuse University in 1975 and has worked with a variety of companies on medical imaging, machine vision, prepress, and digital video. In 1991 Paul collaborated with the Associated Press on developing an electronic darkroom. During the Gulf War, newspapers used his software to process photographs taken of the nighttime attack on Baghdad by the United States. Working with Digital, he has contributed to adding video to the Alpha desktop.

Robert A. Ulichney
Robert Ulichney received a Ph.D. from the Massachusetts Institute of Technology in electrical engineering and computer science and a B.S. in physics and computer science from the University of Dayton, Ohio. He joined Digital in 1978. He is currently a senior consulting engineer with Digital's Cambridge Research Laboratory, where he leads the Video and Image Processing project. He has filed several patents for contributions to Digital products in the areas of hardware and software-only motion video, graphics controllers, and hard copy. Bob is the author of *Digital Halftoning* and serves as a referee for a number of technical societies, including IEEE, of which he is a senior member.

Larry D. Seiler
Robert A. Ulichney

Integrating Video Rendering into Graphics Accelerator Chips

The fusion of multimedia and traditional computer graphics has long been predicted but has been slow to happen. The delay is due to many factors, including their dramatically different data type and bandwidth requirements. Digital has designed a pair of related graphics accelerator chips that integrate video rendering primitives with two-dimensional and three-dimensional synthetic graphics primitives. The chips perform one-dimensional filtering and scaling on either YUV or RGB source data. One implementation dithers YUV source data down to 256 colors. The other converts YUV to 24-bit RGB, which is then optionally dithered. Both chips leave image decompression to the CPU. The result is significantly faster frame rates at higher video quality, especially for displaying enlarged images. The paper compares the implementation cost of various design alternatives and presents performance comparisons with software image rendering.

For years, the computer industry confidently predicted that ubiquitous, integrated multimedia computing was just around the corner. After a number of delays, this computing environment is finally a reality. It is now possible to buy personal computers (PCs) and workstations that combine audio processing with real-time display and manipulation of video or other sampled data, though usually with significant limitations.

For the most part, the industry has followed one of two paths to achieve real-time video processing. On one path, video features are implemented almost entirely in software. When applied to the display of moving images, this approach typically results in a combination of low resolution, slow update times, and small images.

The alternative has been to achieve good video image display performance by adding a separate video hardware option to a PC. Image display is integrated in the box and on the screen but is distinct from the hardware that implements traditional synthetic graphics. Frequently, this design forces performance compromises, for example, by limiting the number of video images that can appear at the same time or by limiting the interaction of images with the window system.

Recently, two key enabling technologies have combined to make a better solution possible. Advances in silicon technology enable low-cost graphics controller chips to be designed with a significant number of gates dedicated to supporting multimedia features. In addition, the peripheral component interconnect (PCI) bus provides high-bandwidth, peer-to-peer communication between the CPU, the main memory, and option cards. Peak bandwidth on the standard 32-bit PCI bus is 133 megabytes per second (MB/s), and higher-performance versions are also available. Good PCI implementations can transfer sequential data at 80 to 100 MB/s. Equally important, the PCI bus allows multimedia solutions to be incrementally built up from a software-only implementation through various levels of hardware support. The *PCI Multimedia Design Guide* describes this incremental approach and also provides standards for latency and video data formats.[1]

This paper describes a Digital engineering project whose goal was to combine video rendering features and traditional synthetic graphics into a unified graphics chip, yielding high-quality, real-time image display as part of the base graphics option at minimal extra cost. This project resulted in two chip implementations, each with its own variation of the same basic design. The TGA2 chip was designed in the Worksystems Group for use in Digital's PowerStorm 3D30 and PowerStorm 4D20 graphics options. The Dagger chip (DECchip 21130) was designed in the Silicon Engineering Group to match the needs of the PC market. The TGA2 and Dagger chips are PCI bus masters and can accept video data from either the host CPU or other video hardware on the PCI bus.

The basic block diagram of the two chips is illustrated in Figure 1. PCI commands are interpreted as either direct memory access (DMA) requests or drawing commands, which the pixel engine block converts to frame buffer read and write operations. Alternately, PCI commands can directly access the frame buffer or the video graphics array (VGA) and RAMDAC logic. In the Dagger chip, the VGA and RAMDAC logic is on-chip; in the TGA2 chip, this logic is implemented off-chip. Most of the video rendering logic is contained in the pixel engine block; the command interpreter and DMA engine blocks require some additional logic to support video rendering.

The following sections describe the capabilities, costs, and trade-offs of the video rendering feature set as implemented in the Dagger and TGA2 graphics chips.

Defining a Low-level Video Rendering Feature Set

The key question when integrating multimedia into a traditional synthetic graphics chip is which features should be implemented in hardware and which should be left in software. A cost-effective design cannot include enough gates to implement every feature of interest. In addition, time-to-market concerns do not allow all features to be designed into the hardware. Therefore, it is essential for designers to define the primary trade-off between features that can be easily and effectively implemented in hardware and those that can be more easily implemented in software without compromising performance.

For the Dagger and TGA2 graphics chips, our basic decision was to leave image compression and decompression in software and put all pixel processing operations into hardware. This approach lets software do what it does best, which is perform complex control of relatively small amounts of data. It also lets hardware do what it does best, which is process large amounts of data where the control is relatively simple and is independent of the data. Specifically, in these two graphics chips, image scaling, filtering, and pixel format conversions are all performed in hardware.

Performing the scaling in hardware greatly reduces the amount of data that the software must process and that must be transmitted over the PCI bus. For example, a 320-by-240-pixel image represented with 16-bit pixels requires just 150K bytes. Even at 30 frames per second (fps), transmitting an image of this size consumes about 5 percent of the available bandwidth of a good PCI bus implementation. This data could be displayed as a 1,280 by 960 array of 32-bit pixels for display, which would use more than 80 percent of the PCI bus bandwidth, if the scaling and pixel format conversion occurs in software.

One data-intensive operation that we chose not to implement in hardware is video input. Designers will need to revisit this decision with each new generation

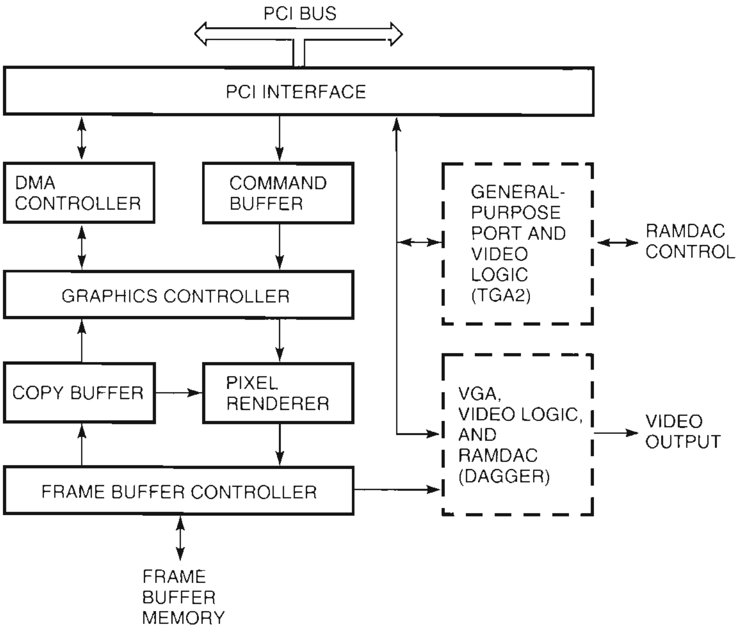


Figure 1
Dagger and TGA2 Chip Structure

of graphics chips. For the current generation, we decided to require the use of a separate video input card for the subset of systems that require video capture. We decided not to include video capture support in the Dagger and TGA2 chips for two basic reasons. First, current application-specific integrated circuit (ASIC) technology would have allowed only a partial solution. We could have put a video input port in hardware but could not have supported the complex operations needed for image compression.

The second reason stems from a market issue. Video display is rapidly becoming ubiquitous, just as mice and multiwindow displays have become commonplace for interacting with PCs and workstations. It is now practical to support high-quality, real-time video display in the base graphics chip. However, the market for video input stations is still much smaller than the market for video display stations. When the size of the video input station market is large enough, and the cost of integrating video input is small enough, support for video input should be added to the base graphics chip.

Video Rendering Pipeline

This section describes the stages of video rendering that are implemented in the Dagger and TGA2 graphics chips. These stages are pixel preprocessing, scaling and filtering, dithering, and color conversion. In some cases, such as scaling and filtering, the two implementations are practically identical. In others, such as color conversion, dramatically different implementations are used to address the differences in requirements for the two chips.

Pixel Preprocessing

The first stage in the pipeline inputs pixel data and converts it into a standard form to be used by the rest of the pipeline. This involves both converting input pixels to a standard format and pretranslating pixel values or color component values. The Dagger and TGA2 chips use DMA over the PCI bus to read packed arrays of pixels from memory.

Pixel Format Conversion Multimedia images are typically represented in YUV format, where the *Y* channel specifies luminance and the *U* and *V* channels represent chrominance. After the CPU has decompressed the source image into arrays of *Y*, *U*, and *V* pixel values, this data is transmitted to the graphics chip in one of a number of standard formats. Alternately, images may be specified as red/green/blue (RGB) triples instead of YUV triples, or as a single index value that specifies a color from a color map random-access memory (RAM) in the video logic. The *PCI Multimedia Design Guide* specifies many standard pixel formats.[1]

Figure 2 shows some of the input pixel formats that are supported in the Dagger and TGA2 graphics chips. The YUV formats on the left allocate 8 bits for each channel. The upper format of the four uses 32 bits per YUV pixel and is called YUV-4:4:4+α.[1] The alpha field is optional and is not used in the Dagger and TGA2 chips. Alpha values are used for blending operations with partially transparent pixels. An alpha value of zero represents a fully transparent pixel, and the maximum value represents a fully opaque pixel.

The remaining three YUV formats specify a separate *Y* value per pixel but subsample the *U* and *V* values so that a pair of pixels shares the same *U* and *V* values. Most YUV compression schemes subsample the chrominance channels, so this approach does not represent any loss of data from the decompressed image. Since the human visual system is more sensitive to changes in luminance than to changes in chrominance, for natural images, *U* and *V* can be subsampled with little loss of image quality.

The three 16-bit YUV formats represent the most common orderings for chrominance-subsampled YUV values. The little-endian and gib-endian orderings are called YUV-4:2:2.[1] The little-endian ordering is the order that is typically produced on the PCI bus

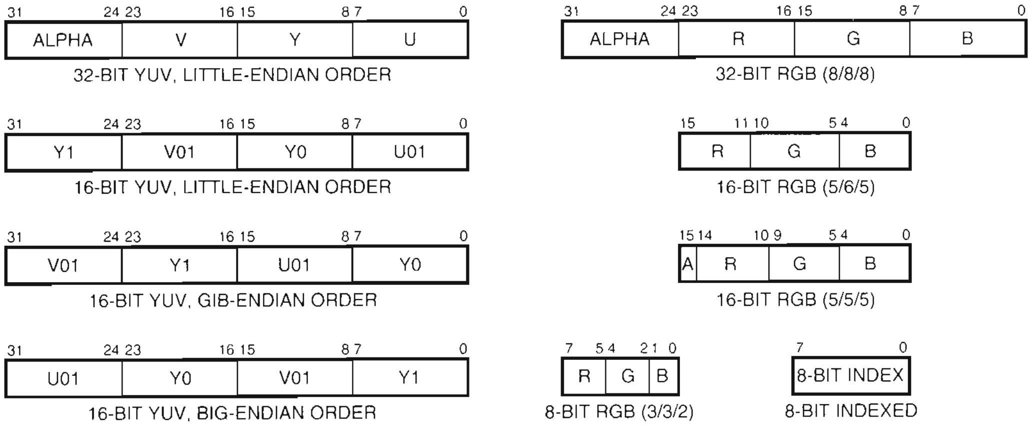


Figure 2
YUV and RGB Pixel Formats in the Dagger and TGA2 Chips

by a little-endian machine. The gib-endian ordering is produced on the PCI bus by a big-endian machine that converts its data to little-endian order, as required for transfer across the PCI bus. That operation preserves byte order for 8-bit and 32-bit data types but not for 16-bit data types like this one. Finally, the big-endian byte ordering is used by some video rendering software and hardware options.

The RGB formats on the right side of Figure 2 allocate varying numbers of bits to the red, green, and blue color channels to produce 8-bit to 32-bit pixels. To achieve acceptable appearance, 8-bit RGB requires high-quality dithering, such as that provided by the AccuVideo dithering technology contained in the Dagger and TGA2 chips and described later in this section. Thirty-two-bit RGB has an optional alpha channel that is not used in the Dagger and TGA2 chips. Some hardware uses the field for control bits or overlay planes instead of for the alpha value. Two different 16-bit RGB formats are common. One format provides 5 bits per color channel and a single alpha bit that indicates transparent or opaque. The other format provides an extra bit for the green channel, since the eye is more sensitive to green than to red or blue.

Finally, 8-bit indexed format is shown at the bottom of Figure 2. This format is simply an 8-bit value that represents an index into a color map. Dagger has an integral color map and digital-to-analog converter, whereas TGA2 requires an external RAMDAC chip to provide its color map. The 8-bit indexed format can represent an indexed range of values or simply a collection of independent values, depending on the needs of the application. In the Dagger and TGA2 chips, the 8-bit indexed format is processed by being passed through the *Y* channel.

Once in the pipeline, the pixels are converted to a standard format consisting of three 8-bit values per pixel. The three values represent RGB or YUV components, depending on the original pixel format. If the original field contains fewer than 8 bits, for example, in the 8-bit RGB format, then the available bits are replicated. Figure 3 shows the expansion of RGB pixels to 8/8/8 RGB format. Replicating the available bits to fill low-order bit positions is preferable to filling the low-order bits with zeros, since replication stretches out the original range of values to include both the lowest and highest values in the 8-bit range, with roughly equal steps between them.

Adjust Look-up Table In the TGA2 chip, a 256-entry look-up table (LUT) may be used during pixel preprocessing. Figure 7 (discussed in the section Color Conversion Algorithms) shows this table, called the adjust LUT, in the TGA2 pipeline. This table supports two different data conversions: luminance adjustment and color index conversion. The adjust LUT is not available in the Dagger chip because it requires too many gates to meet the chip cost goal for Dagger.

Luminance adjustment is used with YUV pixel formats. When this feature is selected, the 8-bit *Y* value from the input pixel is used as an index into the adjust LUT. The 8-bit value read from the table is used as *Y* in the next pipeline stage. Proper programming of the table allows arbitrary luminance adjustment functions to be performed on the input *Y* value; brightness and contrast control are typically provided through this mechanism. Standards for digitally encoding video specify limited ranges for the *Y, U,* and *V* values, largely to prevent analog noise from creating out-of-range values.[2] A particularly important use of this luminance-adjust feature is correcting the poor contrast that would otherwise result from this range limitation. In this case, the adjust LUT may be used to remap the *Y* values to cover the full range of values from 0 to 255.

Another desirable feature is chrominance adjustment, under which the *U* and *V* values are also arbitrarily remapped. The J300 provides this feature; however, TGA2 does not, for two reasons.[3] First, chrominance adjustment is required less often than luminance adjustment and can be emulated in software when the feature is required. Second, chrominance adjustment consumes a significant amount of chip area—either 2K or 4K bits of memory, depending on whether *U* and *V* use the same table or different tables. In this generation of graphics chips, the feature could not be justified in the TGA2 chip. The Dagger chip, which was

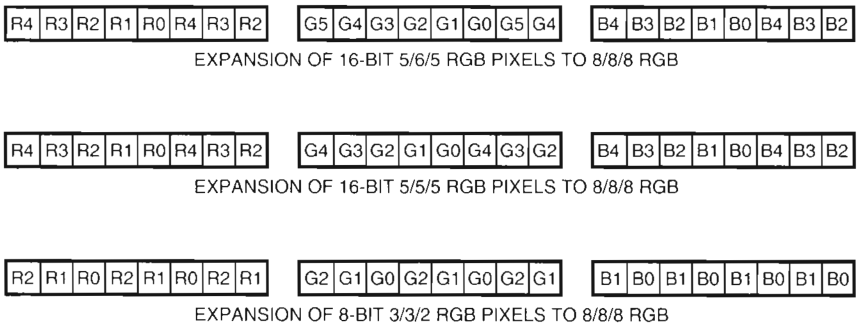


Figure 3
Expanding RGB Pixels to 8/8/8 RGB Format

intended for lower-cost systems, includes neither chrominance nor luminance adjust LUTs.

The other use for the adjust LUT in the TGA2 chip is for color index conversion. This operation can be performed when the input pixel format is 8 bits wide. In this case, the 8-bit input pixel is used as an index into the table. The resulting value is used as the *Y*-channel value in the rest of the pipeline, and the *U* and *V* channels are ignored. Later in the pipeline, the color conversion stage is skipped, and the *Y*-channel value is used directly as the resulting 8-bit pixel value.

Color index conversion is an operation that is particularly desirable when using the Windows NT operating system. Typically, 8-bit screen pixels are converted to displayed colors by means of a color LUT in the back-end video logic. Under the X Window System graphical windowing environment, the mapping between an index and its color can be changed only by the application. Under the Windows NT operating system, however, the mappings may change dynamically. Therefore, an application that has stored an image as 8-bit index values will need to remap those index values before copying it to the screen. This conversion can be done in software, but it is faster and simpler to use the adjust LUT in the TGA2 chip to perform the remapping.

Scaling and Filtering

In the next stage in the rendering pipeline, the chip performs scaling and filtering. The Dagger and TGA2 chips support one-dimensional (1-D) scaling and filtering in hardware. Limiting the chips to 1-D filtering significantly simplifies the chip logic, since no line buffers are needed. Somewhat higher-quality images can be achieved using two-dimensional (2-D) filtering, but the difference is not significant. This difference is further reduced by the AccuVideo dithering algorithm that is implemented by the Dagger and TGA2 chips. Two-dimensional smoothing filters can be supported with added software processing, if required.

Bresenham-style Scaling Image scaling in the Dagger and TGA2 chips uses pixel replication but is not limited to integer multiples. Instead, images can be scaled from any integral source width to any integral destination width. Scaling is implemented through an adaptation of the Bresenham line-drawing algorithm. A complete description of this Bresenham-style scaling algorithm appears in "Bresenham-style Scaling"; the following paragraphs provide an outline of the algorithm, which is the same scaling algorithm used in the J300 family of adapters.[3,4]

The Bresenham scaling algorithm works like the Bresenham line-drawing algorithm. Suppose we are drawing a line from (0, 0) to (10, 6), so that $dx = 10$ and $dy = 6$. This is an *X*-major line; that is, the line is longer in the *X* dimension than in the *Y* dimension. The Bresenham algorithm draws this vector by initializing an error term and then incrementing it *dx* times, in this example, 10 times. Each time the algorithm increments the term, a pixel is drawn. The sign of the error term determines whether to find the next pixel position by stepping to the right (incrementing the *X* position) or by stepping diagonally (incrementing both *X* and *Y*). The error term is incremented in such a way that as the *X* position is incremented 10 times, the *Y* position is incremented 6 times, thus drawing the desired vector.

For Bresenham scaling, *dx* represents the width of the source image, and *dy* represents the width of the destination image on the screen. When reducing the size of the source image, *dx* is greater than *dy* and the error terms and increments are set up in the same way as the *X*-major Bresenham line drawing, as described in the previous paragraph. One source pixel is processed each time the error term is incremented. When Bresenham's line algorithm indicates a step in the *X* dimension only, the source pixel is skipped. When the algorithm indicates a step in both the *X* and the *Y* dimensions, the source pixel is written to the destination. As a result, exactly *dx* source pixels are processed, and exactly *dy* of them are drawn to the screen.

Enlarging an image works in a similar fashion. For example, consider a source image that is narrower than the destination image, that is, *dx* is less than *dy*. This is equivalent to drawing a *Y*-major Bresenham line in which the error term is incremented *dy* times and the *X* dimension is incremented *dx* times. The scaling algorithm draws a source pixel to the destination at each step. If the line-drawing algorithm increments only in the *Y* dimension, it repeats the current pixel. If the line-drawing algorithm increments in both the *X* and the *Y* dimensions, it steps to and displays the next source pixel. Consequently, the *dx* source pixels are replicated to yield *dy* destination pixels, thus enlarging the image.

The Bresenham line-drawing algorithm has two nice properties that are shared by the Bresenham scaling algorithm. First, it requires no divisions to compute the error increments. Second, it produces lines that are as smooth as possible, given the pixel grid. That is, for an *X*-major line, each of the *dx* pixels has a *Y* position that is the closest pixel to the intersection of its *X* position with the real vector. Similarly, the Bresenham scaling algorithm selects pixels that have the most even spacing possible, given the pixel grid.

Just as lines can be drawn from left to right or from right to left, images can be drawn in either direction. An image drawn in one direction is the mirror image of the image drawn in the other direction. Mirror imaging is sometimes used in teleconferencing, so that users can look at themselves the way they normally see themselves. Similarly, images can be turned upside down by simply drawing to the display from bottom to top instead of from top to bottom.

Scaling in the *Y* dimension is performed similarly to *X*-dimension scaling. On the TGA2 chip, scaling is performed in software instead of in hardware: the software increments an error term to decide whether to skip lines (for reducing) or repeat lines (for enlarging). This is acceptable because the CPU has plenty of spare cycles to perform the scaling computations while the algorithm draws the preceding line. The Dagger chip supports *Y*-dimension scaling in hardware to reduce the number of commands that are needed to scale an image.

Smoothing and Sharpening Filters Like the J300, the Dagger and TGA2 chips provide both smoothing and sharpening filters. Table 1 shows the available filters. All are three-tap filters that are inexpensive to implement in hardware. The smoothing filters are used to improve the quality of scaled images. The sharpening filters provide edge enhancement. The two filters marked with asterisks (*) are available only on the TGA2 chip. The others are available on both the Dagger and the TGA2 chips.

The three rows of Table 1 show three levels of smoothing and sharpening filters that can be applied. The degree of smoothing and sharpening may be selected separately. The first row shows the identify filter. This is selected to disable smoothing or sharpening. The second and third rows show three-tap filters that perform a moderate and an aggressive degree of smoothing or sharpening.

Note that when using the aggressive smoothing filter, the center element does not contribute to the result. This filter is intended for postenlargement smoothing when the scale factor is large. Since enlargement is performed by replicating some of the pixels, the center of any span of three pixels will be identical to one of its neighbors when scaling up by a factor of two or more. As a result, the center pixel affects the resulting image, since it is replicated either to the left or to the right. The (1/2, 0, 1/2) filter affords the greatest degree of smoothing that can be achieved with a three-tap filter.

These filter functions are simple to implement in hardware. The implementation requires storing only the two preceding pixels and performing from one to three addition or subtraction operations. The sharpening filters require an additional clamping step to ensure that the result is in the range 0 to 1. Better filtering functions could be obtained by using five taps instead of three taps but only by significantly increasing the logic required for filtering.

Table 1
Smoothing and Sharpening Filters

Smoothing Filter	Degree of Filtering	Sharpening Filter
(0, 1, 0)	Unfiltered	(0, 1, 0)
(¼, ½, ¼)*	Moderate	(−½, 2, −½)
(½, 0, ½)	Aggressive	(−1, 3, −1)*

* Available only on the TGA2 chip

Pre- and Postfiltering The order in which filters are applied depends on whether the image is being enlarged or reduced. When reducing an image, the Bresenham scaling algorithm eliminates pixels from the source image. This can result in severe aliasing artifacts unless a smoothing filter is applied before scaling. The smoothing filter spreads out the contribution of each source pixel to adjacent source pixels.

When enlarging an image, the smoothing filter is applied after scaling. This smoothes out the edges between replicated blocks of pixels. The smoothing filters eliminate the block effect entirely when enlarging up to two times the source image size. The AccuVideo dithering algorithm also contributes to smoothing out the edges between blocks. Another way to smooth out the edges is to use higher-order interpolation to find destination pixel values. Such methods require more logic and do not necessarily produce a better-looking result, particularly for modest scale factors.

If sharpening or edge enhancement is desired, a sharpening filter is used in addition to whatever smoothing filter is selected. For reducing an image, the sharpening filter is applied after scaling—sharpening an image before reducing its size would only exaggerate aliasing effects. For enlarging an image, the sharpening filter is applied before scaling—sharpening an image after enlarging its size would only amplify the edges between blocks. As a result, when both sharpening and smoothing filters are used, one is applied before scaling and the other is applied after scaling.

AccuVideo Dithering Algorithm

AccuVideo dithering technology is Digital's proprietary high-quality, highly efficient method of rendering video with an arbitrary number of available colors. Included is YUV-to-RGB conversion, if necessary, with careful out-of-bounds color mapping. The general algorithm is described in two other papers in this issue of the *Journal*, which discuss the implementation of the J300 video adapter and software-only video players.[3,5] In the chips described in this paper, we simplified the general implementation of the AccuVideo technology by setting constraints on the number of available colors.

Review of the Basic Algorithm The development of the general mean-preserving multilevel dithering algorithm is presented in "Video Rendering," which appears in an earlier issue of the *Journal*.[6] Figure 4 illustrates the theoretical development of the fundamental algorithm for dithering a simple component of a color image. As stated in the earlier paper,

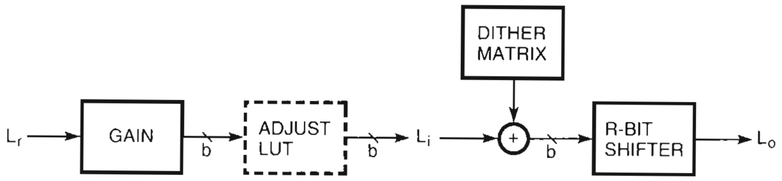


Figure 4
Multilevel Dithering Algorithm Used in the J300, with the Gain Function Separated from the Adjust LUT

a mean-preserving dithered output level L_o can be produced by quantizing the sum of an element from a normalized dither array and an input level L_i by simply shifting the sum to the right by R bits. This simplified quantizer, that is, a quantizer with step size $\Delta_Q = 2^R$, is possible only if the range of input to the adder L_i, or the number of input levels N_i, is properly scaled by a gain G. In the J300 and software-only implementations, G is included in an adjust LUT. In Figure 4, we explicitly separate G from the adjust LUT. The adjust LUT is optionally used to control characteristics such as contrast, brightness, and saturation.

The components of this dithering system can be designed by specifying three parameters:

1. N_r, the number of raw input levels of the given color component
2. N_o, the number of desired output levels
3. b, the width of the adder in bits, and the number of bits used to represent the input levels

Using the results from the multilevel dithering algorithm, the number of bits to be right-shifted is

$$R = \text{int}\left\{ \log_2\left(\frac{2^b - 1}{N_o - 1}\right)\right\}$$

and the gain is

$$G = \frac{N_i - 1}{N_r - 1},$$

where

$$N_i = (N_o - 1)2^R + 1.$$

The effect of the gain is multiplicative. That is, $L_i = L_r \times G$, where L_r is the raw input level. In the absence of an adjust LUT, this multiplication must be explicitly performed.

Simplified Implementation of Gain In the above summary of the basic dithering algorithm, the values of N_r and N_o can be any integer, where $N_r > N_o$. Consider the important special case of restricting these values to be powers of two. Introducing the three integers p, q, and z, we specify that $N_r = 2^p$, $N_o = 2^q$, and $b = p + z$, where z is the number of additional bits used to represent L_i over L_r. $z > 0$ guarantees that $N_i > N_r$, thus ensuring that all the raw input levels will be distinguished by the dithering system. $z = 0$ causes $N_i < N_r$. This situation results in some loss of resolution of raw input levels, because, in all cases, the number of perceived output levels from the dithering system will be at most N_i.

Using this information and the expressions of R and G, it is straightforward to show that $R = p - q + z$, and

$$N_i = (2^q - 1)2^R + 1.$$

Further,

$$G = \frac{((2^q - 1)2^R + 1) - 1}{2^p - 1} = \frac{(2^q - 1)2^p}{(2^p - 1)2^{(q-z)}}.$$

A key approximation made at this point is

$$\frac{2^p}{2^p - 1} \approx 1.$$

Note that this approximation becomes better as the number of bits, p, in the raw input increases.

An approximate gain thus simplifies to

$$\hat{G} = \frac{(2^q - 1)}{2^{(q-z)}} = 2^z - \frac{1}{2^{(q-z)}}.$$

With this value of $\hat{G}$, the resulting modified input levels will be proportionally less than ideal by a factor of

$$\frac{\hat{G}}{G} = \frac{2^p - 1}{2^p}.$$

The fact that this error is negative guarantees that overflow will never occur in the multilevel dithering system. Therefore, a truncation step is not needed in the implementation. Figure 5 illustrates the implementation of $\hat{G}$, which consists of the subtraction of a $(q - z)$-bit right shift of L_r from a z-bit left shift of L_r. This simple "multiplier" is what is implemented in Dagger, TGA2, and the ZLX family of graphics accelerators, where the power-of-two constraint on the output levels is made.

Consider, for example, the case where $p = 8$ ($N_r = 256$), $q = 3$ ($N_o = 8$), and $z = 1$. From the equations just presented, $R = 6$, $b = 9$, and $N_i = 449$. Although our approximation for the gain, $\hat{G} = (2 - 1/4) = 1.75$,

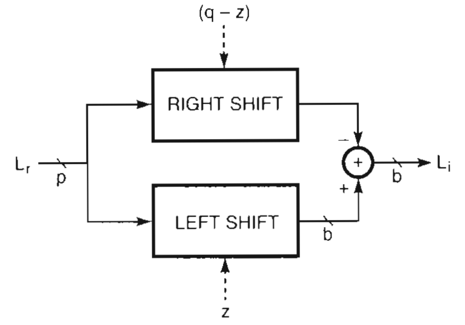


Figure 5
Parallel-shifter Implementation of the Gain Function

is not equal to the ideal gain, $G = 448/255 \approx 1.757$, the ratio $\hat{G}/G \approx 0.996$ is so close to unity that any resulting differences in output are indistinguishable.

Shared Dither Matrix Another simplification can be made by having all the color components in the rendering system share the same dither matrix. As defined in "Video Rendering," a dither template is an array of N_t unique elements, with values $T \in \{0, 1, \ldots, (N_t - 1)\}$.[6] These elements are normalized to establish the dither matrix element d for each location $[x, y]$ as follows:

$$d[x,y] = \text{int}\left\{ \frac{2^R}{N_t}\left(T[x,y] + \frac{1}{2} \right) \right\}.$$

For any real number A and any positive integer K, the following is always true:

$$\text{int}\left\{ \frac{A}{K} \right\} = \text{int}\left\{ \frac{\text{int}\, A}{K} \right\}.$$

If, for each color component, N_o is a power of two, we can exploit this fact by storing only a single dither matrix designed for the smallest value of N_o. Specifically, this would be $N_o = 2^{(b - R_m)}$, where b is the width in bits of the adder and R_m is the largest value of R in the system. For the other larger number of output levels $N_o{}' = 2^{(b - R')}$ with smaller values of R', normalized dither matrix values $d'[x, y]$ can easily be derived by a simple right shift by $(R_m - R')$ bits of the stored dither matrix, as shown in the following equation:

$$d'[x,y] = \text{int}\left\{ \frac{d[x,y]}{2^{R_m - R'}} \right\}.$$

Since our dither matrices are typically 32 by 32 in size, the hardware savings in storing only one matrix is significant. Also, the stored values can be read-only memory (ROM) instead of the more costly RAM. Typically, RAM requires up to eight times the area of ROM in either gate array or custom implementations.

Color Conversion Algorithms

The result of the preceding pipeline stages is three 8-bit values that represent either RGB or YUV color channels. If this format is to be written to the frame buffer, then no further processing is necessary. If a different destination format is specified, then Dagger and TGA2 must perform a color format conversion. Both chips use the same algorithm to dither RGB values down to a smaller number of bits per color channel. Both chips allow writing YUV pixels to the frame buffer, although TGA2 allows the writing of only the 32-bit YUV format. Finally, both chips can convert YUV pixels into the RGB color space, but they use markedly different algorithms to perform this conversion.

Although YUV pixels can be written to the frame buffer in both Dagger and (to a more limited extent) TGA2, neither chip supports displaying YUV pixels to the screen. YUV pixels may be stored only in the off-screen portion of the frame buffer as intermediate values for further processing. This is because it is far more efficient to convert YUV to RGB in the rendering stage than to perform the conversion in the back-end video logic. At the rendering stage, it need only be done at the image update rate of up to 30 fps. If performed in the back-end video logic, the YUV-to-RGB conversion must also be performed at the screen update rate of up to 76 fps. This extra, higher-speed logic may be justified if preconverting YUV to RGB noticeably reduces the image quality. Given the AccuVideo dithering algorithm, however, postconversion is not necessary.

RGB-to-RGB Color Conversion Even if both the source and the destination pixel formats represent RGB color channels, it may still be necessary to perform a bit-depth conversion. Input pixels are expanded out to 8 bits per color channel for processing through the video rendering pipeline. Destination pixels may have 8, 15, 16, or 24 bits for RGB and so may need to be dithered down to a smaller number of bits per pixel. TGA2 also supports 12-bit RGB, as described later in this section.

Dagger and TGA2 differ somewhat in the specific formats that they support. Dagger allows writes to the frame buffer of 3/3/2, 5/5/5, 5/6/5, and 8/8/8 RGB pixel formats. TGA2 supports all these as source pixels but does not allow writes of 5/5/5 and 5/6/5 RGB, because TGA2 does not support 16-bit pixels in the frame buffer. Dagger supports 16-bit pixels because they are very common in the PC industry. In the workstation industry, however, which is TGA2's market, 16-bit pixels are almost unknown. As the Windows NT operating system gains in popularity, this situation is likely to change.

Instead of supporting 16-bit pixels, TGA2 allows writes to the frame buffer of 4/4/4 RGB pixels, with 16 possible shades for each of the red, green, and blue color channels. This is a standard pixel format for workstation graphics, since it allows two RGB buffers to be stored in the space of a 24-bit, 8/8/8 RGB pixel. This in turn allows double buffering, in which one image is drawn while the other image is displayed. Double buffering is essential for animation applications on large screens, since the rendering logic generally cannot repaint the screen fast enough to avoid flicker effects.

YUV-to-RGB Color Conversion on the Dagger Chip

The key design focus for the Dagger chip was to support low-cost graphics options with the highest possible performance and display quality. As a result, although Dagger supports up to 32 bits per pixel, its design center is for 8-bit-per-pixel displays. Therefore, the algorithm that Dagger uses for converting YUV to RGB produces the best possible results given a limit of just 256 resultant colors.

The resulting dithering system design is shown in Figure 6. Note that the same system is used to dither both RGB data and YUV data. Because the number of output levels for each component is always a power of two, we can use the simple gain circuit of Figure 5 and share the same dither matrix by right-shifting its contents, as derived in the last section. In hardware, this shifting simply requires a multiplexer to select the most significant bits of the data. The dither matrix is 7 bits wide to support dithering down to 2-bit blue values in 3/3/2 RGB, but only 6 dither matrix bits are used for 3-bit output, and only 5 bits are used for 4-bit output.

YUV data is always dithered to 4 bits of *Y* and 3 bits each of *U* and *V*. An additional bit is provided for the *Y* channel because the eye is more sensitive to changes of intensity than to changes of color. These 10 bits are input to a color convert LUT, which is implemented as a ROM. Its contents are generated by an algorithm with some out-of-bounds mapping.[5,7] Approximately three-fourths of the possible combinations of YUV values are outside the range of colors that can be specified in the RGB color space. In these cases, the color convert LUT ROM produces an RGB value that has the same luminance but a less saturated color.

The color convert LUT ROM represents these 256 colors as an 8-bit index that is stored in the frame buffer. One additional bit per pixel in off-screen memory specifies which pixels result from YUV conversion and which are used by other applications. When pixels are read from the frame buffer for display to the screen, Dagger's internal RAMDAC reads that additional bit per pixel to decide whether to map each byte through a standard 256-entry color map or through a ROM that is loaded with the 256 colors selected in the color convert LUT ROM. As a result, Dagger allows selection of the best 256 colors for YUV-to-RGB conversion, in addition to allowing color-mapped applications to store 8-bit index values in the frame buffer.

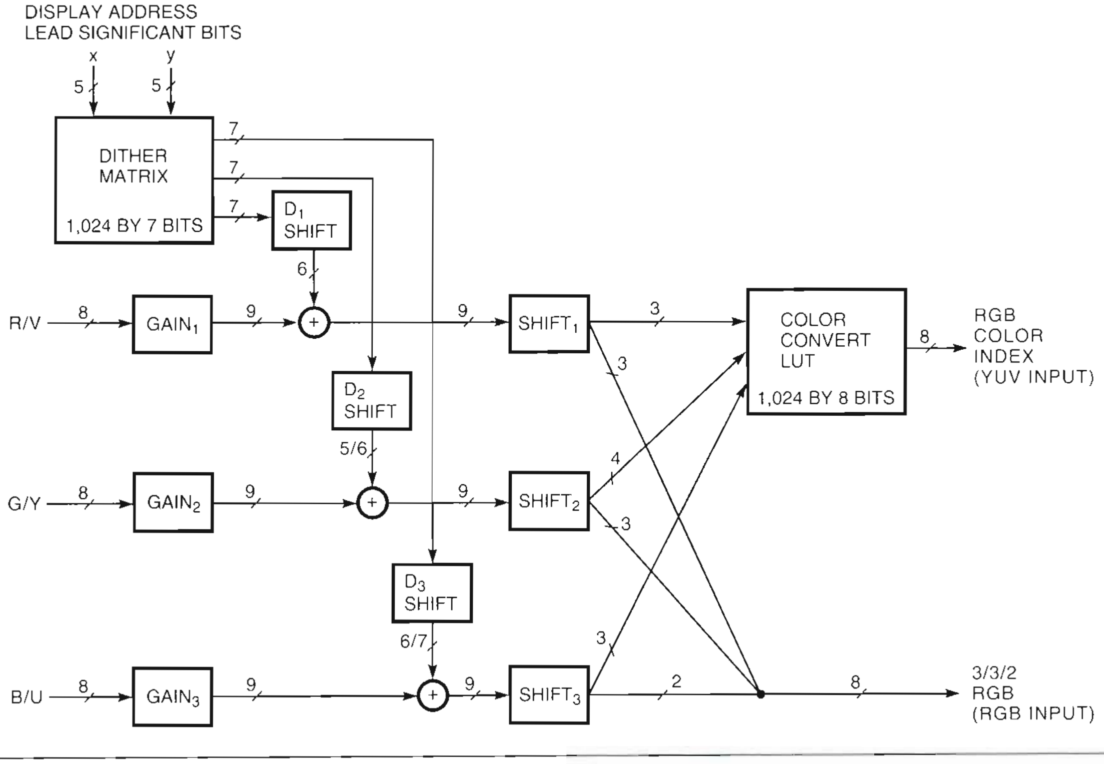


Figure 6
Dithering and YUV-to-RGB Conversion in the Dagger Chip

It is possible to extend this approach to use more bits of dithered YUV to produce more finely quantized RGB colors. The size of the required look-up ROM quickly gets out of hand, however. Dagger uses a 1K-by-8-bit ROM to convert 4/3/3 YUV into 256 RGB colors. Using 4/4/4 YUV would make the ROM five times larger (4K by 10 bits). To produce 4K RGB colors would require a ROM with 16K 12-bit entries.

YUV-to-RGB Color Conversion on TGA2 The TGA2 graphics chip performs dithering and color conversion in the reverse order, as compared to the Dagger chip. In TGA2, a YUV pixel is first converted into an RGB pixel at 8 bits per channel. This 24-bit RGB pixel is then either written to the frame buffer or dithered down to 8- or 12-bit RGB before being written to the frame buffer. Figure 7 shows the dithering system that is used in the TGA2 chip.

The key advantage of the TGA2 approach over the Dagger approach is that it allows deeper frame buffers to use higher-quality color conversion. If a 24-bit frame buffer is being used, TGA2 allows YUV to be converted to full 8/8/8 RGB. On the Dagger chip, YUV-to-RGB conversion produces only 256 different colors, regardless of the frame buffer depth. This is acceptable on Dagger, where 24-bit frame buffers are far from the design center. Also, the Dagger method uses fewer gates, which is an important consideration for the cost-constrained Dagger implementation.

Another advantage of this algorithm for TGA2 is that the set of colors used for video image display is the same one used by full-color synthetic graphics applications, such as a solid modeling package or a scientific visualization application. This allows a common color map to be used by both image applications and shaded graphics applications. Unlike the Dagger chip, TGA2 does not have an integrated RAMDAC and uses an external RAMDAC. Typical low-cost RAMDAC chips provide only one 256-entry color map, so it is important for TGA2 to allow image applications to share this color map with other applications.

Figure 8 illustrates how the TGA2 chip performs YUV-to-RGB color conversion. By the standard definition of the YUV format, the conversion to RGB consists of a 3-by-3 matrix multiplication operation in which three terms equal 1 and two terms equal 0.[2] The TGA2 chip performs this matrix multiplication using four LUTs to perform the remaining four multiplications, together with some adders. A final multiplexer is required to clamp the resulting values to the range 0 to 255.

The TGA2 color conversion algorithm has one disadvantage: the algorithm does not handle out-of-range YUV values as well as the technique used in the Dagger chip. In Dagger, each YUV triple that is out of range has an optimal or near-optimal RGB triple computed for it and placed in the table. With the TGA2 technique, the red, green, and blue components are computed separately. The individual color components are clamped to the range boundaries, but if a YUV triple results in an out-of-range value for green, this cannot affect the red or blue values. The result is some color distortion for oversaturated images. If such a result would be unsatisfactory, it is necessary to adjust the colors in software, e.g., by reducing the saturation or the intensity of the source image so that most YUV triples map to valid RGB colors.

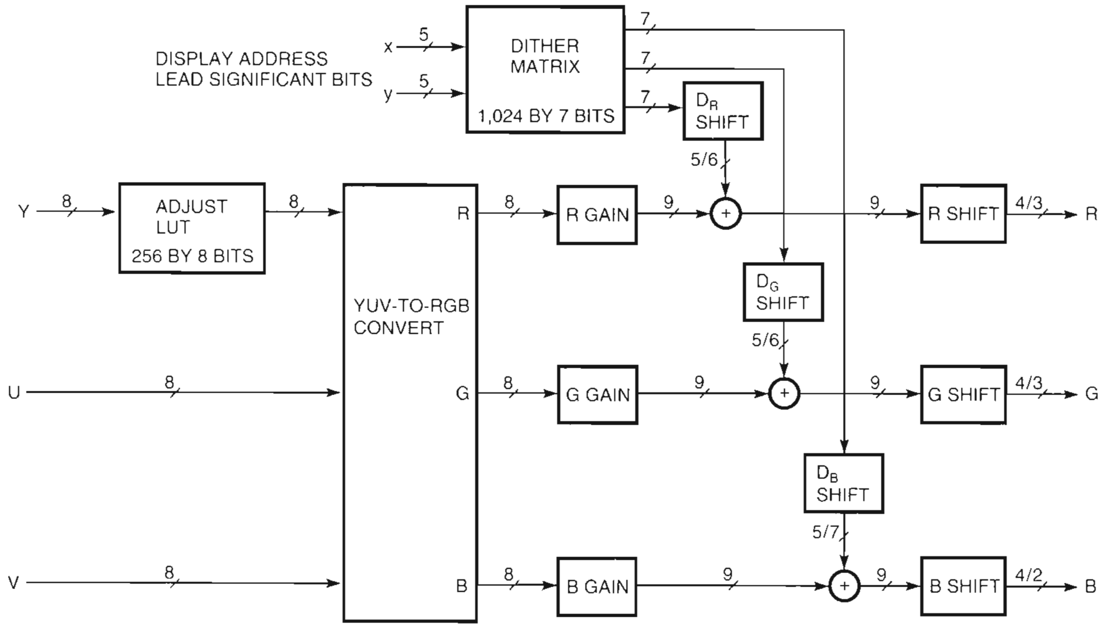


Figure 7
Dithering System in the TGA2 Chip

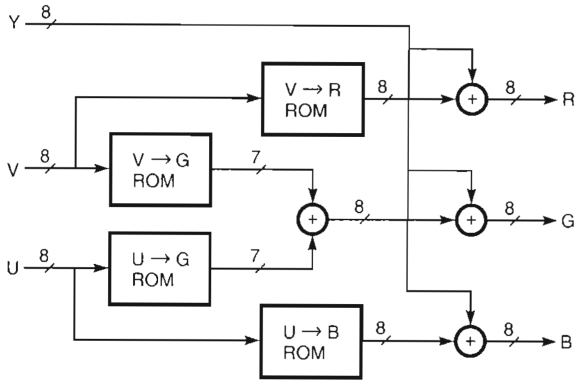


Figure 8
YUV-to-RGB Conversion in the TGA2 Chip

Table 2
Gates Used by the TGA2 Video Rendering Logic

Logic Block	Number of Cells	Number of Gates	Gates per Total Gates (Percent)
Pixel Formatting	778	584	4.2
Look-up Table	9,590	7,192	52.3
Filtering	2,265	1,699	12.4
Color Convert	3,486	2,614	19.0
Miscellaneous	2,210	1,658	12.1
Total	18,329	13,747	100.0

Implementation Cost and Performance

Both the Dagger and the TGA2 chips have the design goal of integrating as many as possible of the J300 design features into a single-chip graphics and video solution. Dagger and TGA2 include different features and implement some common features in different ways because each chip focuses on a different market. As mentioned earlier, Dagger is a PC graphics accelerator chip, and TGA2 is a workstation graphics accelerator chip.

Gate Cost

Table 2 shows the number of gates required to add the various imaging operations to the TGA2 chip. TGA2 is implemented in IBM's 5L standard cell technology. The video rendering logic represents less than 10 percent of the total TGA2 logic. The chip contains no additional gates for video scaling or dithering logic, since nearly all the gates needed to implement those functions are already required in TGA2 to implement Bresenham line drawing and dithering of 3-D shaded objects.

Table 2 clearly shows why the luminance adjust LUT was omitted from Dagger. On the TGA2 chip, the LUT requires more than half the total gates used for multimedia support.

Display Performance

The peak hardware performance for image operations on the TGA2 chip depends primarily on the internal clock rate, which is 60 megahertz (MHz). The TGA2 chip is fully pipelined, so that one pixel is processed on each clock cycle, regardless of the filtering, conversion, or dithering that is required. Reducing the image requires one clock cycle per source pixel. Enlarging the image requires one clock cycle per destination pixel. Actual hardware performance is never quite equal to peak rates, but TGA2 performance approaches peak rates. For example, TGA2's hardware performance limits support rendering a common intermediate format (CIF) image that is scaled up by a factor of three in both dimensions at over 30 fps.

Actual system performance depends on many factors besides hardware performance. Typically, multimedia images are stored and transmitted in compressed form, so that display performance depends on the speed of the decompression hardware or software. "Software-only Compression, Rendering, and Playback of Digital Video" contains tables that show the performance of a variety of AlphaGeneration systems with software-only rendering and with J300 rendering hardware that implements hardware algorithms similar to those in the TGA2 and Dagger chips.[5]

Table 3 shows the results of preliminary tests of TGA2 video display rates on AlphaStation 250 4/166 and AlphaStation 250 4/266 workstations, which use DECchip 21064 CPUs. The table shows performance in frames per second for displaying the standard Motion Picture Experts Group (MPEG) flower garden video clip, comparing performance to software algorithms that use the TGA graphics accelerator. Like TGA2, the TGA chip supports fast image transfers to the frame buffer; however, TGA does not provide any specific logic to accelerate video display.

The first two lines of Table 3 show performance for displaying images at their original size. Allowing TGA2 to convert decompressed YUV pixels to RGB improves performance by 34 to 45 percent, depending on CPU performance. This performance improvement drops to 18 to 25 when data transfer times are included. Possibly, this gap can be reduced by further coding to better overlap data transfer with MPEG decompression. Note that the TGA2 performance can include image filtering and a luminance adjust table lookup at no loss in performance.

The third line of Table 3 shows performance when the video clip is displayed at two times the size in both dimensions. The flower garden movie covers an area of 320 by 240 pixels, which is very small on a 1,280-by-1,024-pixel monitor. Therefore, it is highly desirable to display an enlarged image. In this case, TGA2

Table 3
Frames per Second for Displaying MPEG Flower Garden Video Clip

AlphaStation 250 4/166				AlphaStation 250 4/266		
TGA (fps)	**TGA2 (fps)**	**Increase (Percent)**		**TGA (fps)**	**TGA2 (fps)**	**Increase (Percent)**
24.7	35.8	45	Software decode rate	47.9	64.2	34
23.1	28.9	25	1x video playback rate	44.0	52.1	18
12.7	26.4	108	2x video playback rate	23.1	44.9	95

Source: Tom Morris, Technical Director, Light and Sound Engineering, Digital Equipment Corporation

displays the video clip at twice the speed of the software algorithm that uses the TGA graphics chip. The subjective difference is even greater, since TGA2 applies a smoothing filter to improve the quality of the resulting images. The software algorithm on the TGA chip performs no filtering because this would dramatically reduce chip performance.

The performance data in Table 3 are for displaying 8-bit images to the frame buffer. TGA2 is able to display 24-bit images at the same performance, up to the limit of its frame buffer bandwidth. For the examples in Table 3, TGA2 is able to produce either 8-bit, 12-bit, or 24-bit images at essentially the same performance. Software algorithms would experience a dramatic drop in performance, simply because they would have to process and transfer three times as much data. Therefore, the TGA2 chip allows significantly higher-quality images to be displayed without sacrificing performance.

Conclusions

This paper describes two graphics accelerator chips that integrate a set of image processing operations with traditional synthetic graphics operations. The image operations are carefully chosen to allow significantly higher performance with minimal extra logic; the operations that can be performed in software are left out. Both chips take advantage of the PCI bus to provide the bandwidth necessary for image data transfers.

The Dagger and TGA2 video rendering logic is based on the AccuVideo rendering pipeline as implemented in the J300 family of video and audio adapters.[3] The following restrictions were made to integrate this logic into these graphics chips:

1. Color preprocessing—Eliminate RAM for dynamic chrominance control. For the Dagger chip, also eliminate RAM for dynamic brightness/contrast control.
2. Filtering—Support just one sharpening and one smoothing filter (other than the identity filters) in the Dagger chip. For the TGA2 chip, support just two sharpening and two smoothing filters.
3. Color output—For the Dagger chip, allow only 256 output colors for YUV input [3/3/2 for RGB input]. For the TGA2 chip, support only RGB colors with a power-of-two number of values in each channel.

The quality of the resulting images is excellent. The AccuVideo 32-by-32 void-and-cluster dithering algorithm provides quality similar to error diffusion dithering algorithms.[8] Error diffusion is a technique in which the difference between the desired color and the displayed color at each pixel is used to control dithering decisions at adjacent pixels. Error-diffusion dithering requires considerably more logic than AccuVideo dithering and cannot be used when rendering synthetic graphics.

The high quality of the AccuVideo algorithm is especially important when dithering down to 8-bit pixels (3/3/2 RGB). Even in this extreme case, applying the AccuVideo dithering algorithm results in a slight graininess but few visible dithering artifacts. Applying AccuVideo dithering to 12-bit (4/4/4 RGB) pixels results in screen images that are almost indistinguishable from 24-bit (8/8/8 RGB) pixels.

We plan to continue evaluating new multimedia features for inclusion in our synthetic graphics chips. Areas we are investigating include more elaborate filtering and scaling operations, additional types of color conversion, and inexpensive ways to accelerate the compression/decompression process.

References

1. *PCI Multimedia Design Guide,* rev 1.0 (Portland, Oreg.: PCI Special Interest Group, March 29, 1994).
2. *Encoding Parameters of Digital Television for Studios,* CCIR Report 601-2 (Geneva: International Radio Consultative Committee [CCIR], 1990).
3. 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.
4. R. Ulichney, "Bresenham-style Scaling," *Proceedings of the IS&T Annual Conference* (Cambridge, Mass., 1993): 101–103.

5. 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.

6. R. Ulichney, "Video Rendering," *Digital Technical Journal*, vol. 5, no. 2 (Spring 1993): 9–18.

7. R. Ulichney, "Method and Apparatus for Mapping a Digital Color Image from a First Color Space to a Second Color Space," U.S. Patent 5,233,684 (1993).

8. R. Ulichney, "The Void-and-Cluster Method for Generating Dither Arrays," *IS&T/SPIE Symposium on Electronic Imaging Science and Technology*, San Jose, Calif., vol. 1913 (February 1993): 332–343.

Biographies

Larry D. Seiler
Larry Seiler is a consultant engineer working in Digital's Graphics and Multimedia Group within the Worksystems Business Unit. During his 15 years at Digital, Larry has helped design a variety of graphics products. Most recently, he was the architect for the TGA2 graphics chip that is used in Digital's PowerStorm 3D30 and PowerStorm 4D20 graphics options. Prior to that he architected the SPX series of graphics options for VAX workstations. Larry holds a Ph.D. in computer science from the Massachusetts Institute of Technology, as well as B.S. and M.S. degrees from the California Institute of Technology.

Robert A. Ulichney
Robert Ulichney received a Ph.D. from the Massachusetts Institute of Technology in electrical engineering and computer science and a B.S. in physics and computer science from the University of Dayton, Ohio. He joined Digital in 1978. Bob is currently a senior consulting engineer with Digital's Cambridge Research Laboratory, where he leads the Video and Image Processing project. He has filed several patents for contributions to Digital products in the areas of hardware and software-only motion video, graphics controllers, and hard copy. Bob is the author of *Digital Halftoning* and serves as a referee for a number of technical societies, including IEEE, of which he is a senior member.

Lawrence S. Cohen
John H. Williams

Technical Description of the DECsafe Available Server Environment

The DECsafe Available Server Environment (ASE) was designed to satisfy the high-availability requirements of mission-critical applications running on the Digital UNIX operating system. By supplying failure detection and failover procedures for redundant hardware and software subsystems, ASE provides services that can tolerate a single point of failure. In addition, ASE supports standard SCSI hardware in shared storage configurations. ASE uses several mechanisms to maintain continuous operation and to prevent data corruption resulting from network partitions.

The advent of shared storage interconnect support such as the small computer system interface (SCSI) in the Digital UNIX operating system provided the opportunity to make existing disk-based services more available. Since high availability is an important feature to mission-critical applications such as database and file system servers, we started to explore high-availability solutions for the UNIX operating system environment. The outcome of this effort is the DECsafe Available Server Environment (ASE), an integrated organization of computer systems and external disks connected to one or more shared SCSI buses.

In the first section of this paper, we review the many product requirements that needed to be explored. We then define the ASE concepts. In the next section, we discuss the design of the ASE components. In subsequent sections, we describe some of the issues that needed to be overcome during the product's design and development: relocating client-server applications, event monitoring and notification, network partitioning, and management of available services. Further, we explain how ASE deals with problems concerning multihost SCSI; the cross-organizational logistical issues of developing specialized SCSI hardware and firmware features on high-volume, low-priced standard commodity hardware; and modifications to the Network File System (NFS) to be both highly available and backward compatible.[1]

Requirements of High-availability Software

The availability concept is simple. If two hosts can access the same data and one host fails, the other host should be able to access the data, thus making the applications that use the data more available. This notion of loosely connecting hosts on a shared storage interconnect is called high availability. High availability lies in the middle of the spectrum of availability solutions, somewhere between expensive fault-tolerant systems and a well-managed, relatively inexpensive, single computer system.[2]

By eliminating hardware single points of failure, the environment becomes more available. The goal of the

ASE project was to achieve a product that could be configured for no single point of failure with respect to the availability of services. Thus we designed ASE to detect and dynamically reconfigure around host, storage device, and network failures.

Many requirements influenced the ASE design. The most overriding requirement was to eliminate the possibility for data corruption. Existing single-system applications implicitly assumed that no other instance was running on another node that could also access the same data. If concurrent access did happen, the data would likely be corrupted. Therefore the preeminent challenge for ASE was to ensure that the application was run only once on only one node.

Another requirement of ASE was to use industry-standard storage and interconnects to perform its function. This essentially meant the use of SCSI storage components, and this did pose some challenges for the project. In a later section, we discuss the challenge of ensuring data integrity in a multihosted SCSI environment. Also, the limitation of eight SCSI devices per SCSI storage bus confined the scaling potential of ASE to relatively small environments of two to four nodes.

Less obvious requirements affected the design. ASE would be a layered product with minimal impact on the base operating system. This decision was made for maintainability reasons. This is not to say we did not make changes to the base operating system to support ASE; however, we made changes only when necessary.

ASE was required to support multiple service types (applications). Originally, it was proposed that ASE support only the Network File System (NFS), as does the HANFS product from International Business Machines Corporation.[3] Customers, however, required support for other, primarily database applications as well. As a result, the ASE design had to evolve to be more general with respect to application availability support.

ASE was also required to allow multiple service types to run concurrently on all nodes. Other high-availability products, e.g., Digital's DECsafe Failover and Hewlett-Packard's SwitchOver UX, are "hot-standby" solutions. They require customers to purchase additional systems that could be idle during normal operation. We felt it was important to allow all members of the ASE to run highly available applications as well as the traditional, hot-standby configuration.

The remaining requirement was time to market. IBM's HA/6000 and Sun Microsystems' SPARCcluster1 products were in the market, offering cluster-like high availability. We wanted to bring out ASE quickly and to follow with a true UNIX cluster product.

One last note for readers who might try to compare ASE with the VMScluster, a fully functional cluster product. ASE addresses the availability of single-threaded applications that require access to storage. For example, it does not address parallel applications that might need a distributed lock manager and concurrent access to data. Another effort was started to address the requirements of clusters in the UNIX environment.[4]

ASE Concepts

To understand the description of the ASE design, the reader needs to be familiar with certain availability concepts and terms. In this section, we define the ASE concepts.

Storage Availability Domain

A storage availability domain (SAD) is the collection of nodes that can access common or shared storage devices in an ASE. Figure 1 shows an example of a SAD. The SAD also includes the hardware that connects the nodes such as network devices and the storage interconnects. The network device can be any standard network interface that supports broadcast. This usually implies either Ethernet or a fiber distributed data interface (FDDI). Although the SAD may include many networks, only one is used for communicating the ASE protocols in the version 1.0 product. To remove this single point of failure, future versions of ASE will allow for communication over multiple networks. Other networks can be used by clients to access ASE services. The storage interconnect is either a single-ended or a fast, wide-differential SCSI. The shared devices are SCSI disks or SCSI storage products like HSZ40 controllers.

Symmetric versus Asymmetric SADs

There are many ways a SAD may be configured with respect to nodes and storage. In a symmetric configuration (see Figure 1), all nodes are connected

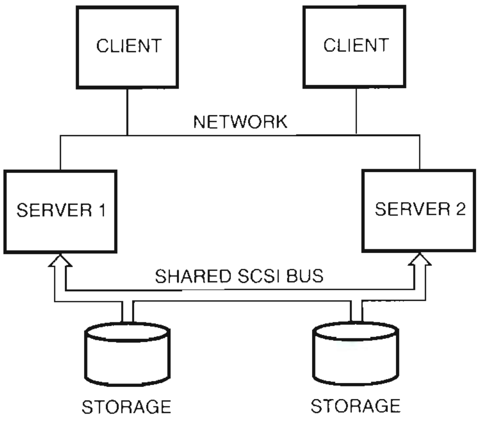


Figure 1
Simple Available Server Environment

to all storage. An asymmetric configuration exists when all nodes are not connected to all the storage devices. Figure 2 shows an asymmetric configuration.

The use of asymmetric configurations improves performance and increases scalability. Performance is better because fewer nodes share the same bus and have less opportunity to saturate a bus with I/O. Scalability is greater because an asymmetric configuration allows for more storage capacity. On the other hand, asymmetric configurations add significant implementation issues that are not present with symmetric configurations. Symmetric configurations allow for simplifying assumptions in device naming, detecting network partitions, and preventing data corruption. By assuming fully connected configurations, we were able to simplify the ASE design and increase the software's reliability. For these reasons, we chose to support only symmetric configurations in version 1.0 of ASE.

Service

We use the term *service* to describe the program (or programs) that is made highly available. The service model provides network access to shared storage through its own client-server protocols. Examples of ASE services are NFS and the ORACLE7 database. Usually, a set of programs or processing steps needs to be executed sequentially to start up or stop the service. If any of the steps cannot be executed successfully, the service either cannot be provided or cannot be stopped. Obviously, if the shared storage is not accessible, the service cannot begin. ASE provides a general infrastructure for specifying the processing steps and the storage dependencies of each service.

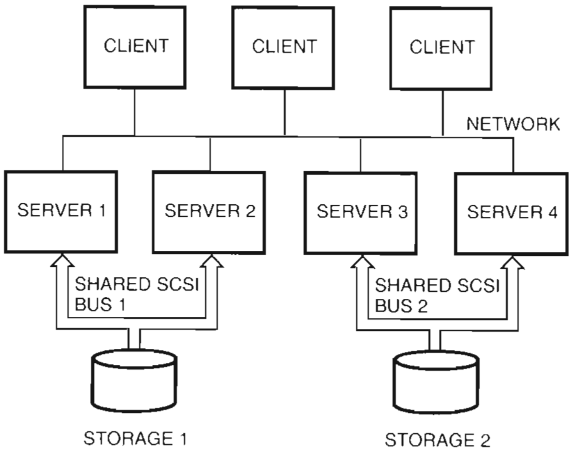


Figure 2
Asymmetric Configuration of ASE

Events and Failure Modes

ASE monitors its hardware and software to determine the status of the environment. A change in status is reported as an event notification to the ASE software. Examples of events include a host failure and recovery, a failed network or disk device, or a command from the ASE management utility.

Service Failover

The ASE software responds to events by relocating services from one node to another. A relocation due to a hardware failure is referred to as *service failover.* There are reasons other than failures to relocate a service. For example, a system manager may relocate a service for load-balancing reasons or may bring down a node to perform maintenance.

Service Relocation Policy

Whenever a service must be relocated, ASE uses configurable policies to determine which node is best suited to run the service. The policy is a function of the event and the installed system-management preferences for each service. For example, a service must be relocated if the node on which it is running goes down or if a SCSI cable is disconnected. The system manager may specify the node to which the service should be relocated. Preferences can also be provided for node recovery behavior. For example, the system manager can specify that a service always returns to a specified node if that node is up. For services that take a long time to start up, the system manager may specify that a service relocate only if its node should fail. Additional service policy choices are built into ASE.

Centralized versus Distributed Control

The ASE software is a collection of daemons (user-level independent processes run in the background) and kernel code that run on all nodes in a SAD. When we were designing the service relocation policy, we could have chosen a distributed design in which the software on each node participated in determining where a service was located. Instead, we chose a centralized design in which only one of the members was responsible for implementing the policy. We preferred a simple design since there was little benefit and much risk to developing a set of complex distributed algorithms.

Detectable Network Partition versus Undetectable Full Partition

A detectable network partition occurs when two or more nodes cannot communicate over their networks but can still access the shared storage. This condition could lead to data corruption if every node reported that all other nodes were down. Each node could try to acquire the service. The service could run concurrently on multiple nodes and possibly corrupt the shared storage. ASE uses several mechanisms to prevent data corruption resulting from network partitions. First, it relies on the ability to communicate status over the SCSI bus. In this way, it can detect network partitions and prevent multiple instances of the service. When communication cannot occur over the SCSI bus, ASE relies on the disjoint electrical connectivity property of the SCSI bus. That is, if Server 1 and Server 2 cannot contact each other on the SCSI bus, it is impossible for both servers to access the same storage on that bus.

As a safeguard to this assumption, ASE also applies device reservations (hard locks) on the disks. The hard lock is an extreme failsafe mechanism that should rarely (if ever) be needed. As a result, ASE is able to adopt a nonquorum approach to network partition handling. In essence, if an application can access the storage it needs to run, it is allowed to run. Quorum approaches require a percentage (usually more than half) of the nodes to be available for proper operation. For two-node configurations, a tiebreaker would be required: if one node failed, the other could continue to operate. In the OpenVMS system, for example, a disk is used as a tiebreaker. We chose the nonquorum approach for ASE because it provides a higher degree of availability.

Although extremely unlikely to occur, there is one situation in which data could become corrupted: a full partition could occur during shadowed storage. Shadowing transparently replicates data on one or more disk storage devices. In a full partition, two nodes cannot communicate via a network, and the SCSI buses are disconnected in a way that the first node sees one set of disks and the second node sees another set. Figure 3 shows an undetectable full partition.

Even though this scenario does not allow for common access to disks, it is possible that storage that is replicated or shadowed across two disks and buses could be corrupted. Each node believes the other is down because there is no communication path. If one node has access to half of the shadowed disk set and the other node has access to the other half, the service may be run on both nodes. The shadowed set would become out of sync, causing data corruption when its halves were merged back together. Because the possibility of getting three faults of this nature is infinitesimal, we provide an optional policy for running a service when less than a full shadowed set is available.

Service Management

ASE service management provides three functions: service setup, SAD monitoring, and service relocation. The management program assists in the creation of services by prompting for information such as the type of service, the disks and file systems that are required, and shadowing requirements. ASE gathers the requirements and creates the command sequences that will start the service. It thus integrates complex subsystems such as the local file systems, logical storage management (LSM), and NFS into a single service.

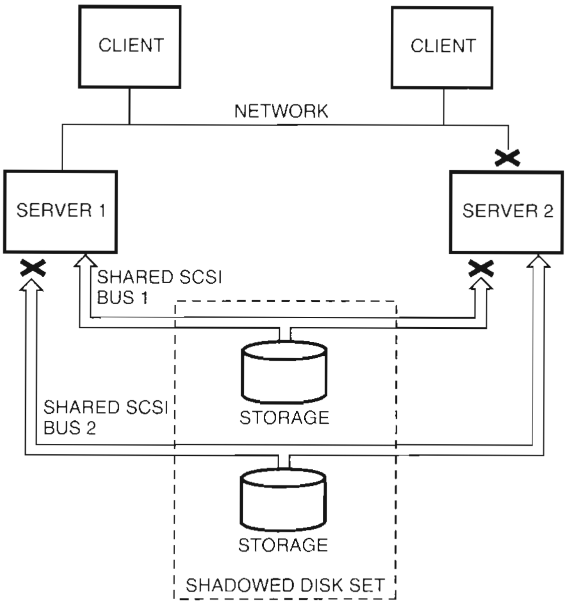


Figure 3
Full Partition

ASE version 1.0 supports three service types: user, disk, and NFS. A *user service* requires no disks and simply allows a user-supplied script to be executed whenever a node goes up or down. The *disk service* is a user service that also requires disk access, that is, disk and file system information. The disk service, for example, would be used for the creation of a highly available database. The *NFS service* is a specialized version of the disk service; it prompts for the additional information that is germane to NFS, for example, export information.

The monitoring feature provides the status of a service, indicating whether the service is running or not and where. It also provides the status of each node.

The service location feature allows system managers to move services manually by simply specifying the new location.

Software Mirroring

Software mirroring (shadowing) is a mechanism to replicate data across two or more disks. If one disk fails, the data is available on another disk. ASE relies on Digital's LSM product to provide this feature.

ASE Component Design

The ASE product components perform distinct operations that correspond to one of the following categories:

1. Configuring the availability environment and services
2. Monitoring the status of the availability environment
3. Controlling and synchronizing service relocation
4. Controlling and performing single-system ASE management operations
5. Logging events for the availability environment

The configuration of ASE is divided into the installation and ongoing configuration tasks. The ASE installation process ensures that all the members are running ASE-compliant kernels and the required daemons (independent processes) for monitoring the environment and performing single-system ASE operations. Figure 4 illustrates these components. The shared networks and distributed time services must also be configured on each member to guarantee connectivity and synchronized time. The most current ASE configuration information is determined from time stamps. Configuration information that uses time stamps does not change often or frequently and is protected by a distributed lock.

The ASE configuration begins by running the ASE administrative command (ASEMGR) to establish the membership list. All the participating hosts and daemons must be available and operational to complete this task successfully. ASE remains in the install state until the membership list has been successfully processed. As part of the ASE membership processing, an ASE configuration database (ASECDB) is created, and the ASE member with the highest Internet Protocol (IP) address on the primary network is designated to run the ASE director daemon (ASEDIRECTOR). The ASE director provides distributed control across the ASE members. Once an ASE director is running, the ASEMGR command is used to configure and control individual services on the ASE members. The ASE agent daemon (ASEAGENT) is responsible for performing all the single-system ASE operations required to manage the ASE and related services. This local system management is usually accomplished by executing scripts in a specific order to control the start, stop, add, delete, or check of a service or set of services.

The ASE director is responsible for controlling and synchronizing the ASE and the available services dependent on the ASE. All distributed decisions are made by the ASE director. It is necessary that only one ASE director be running and controlling an ASE to provide a centralized point of control across the ASE. The ASE director provides the distributed orchestration of service operations to effect the desired recovery or load-balancing scenarios. The ASE director controls the availability services by issuing sets of service actions to the ASE agents running on each member. The ASE director implements all failover strategy and control.

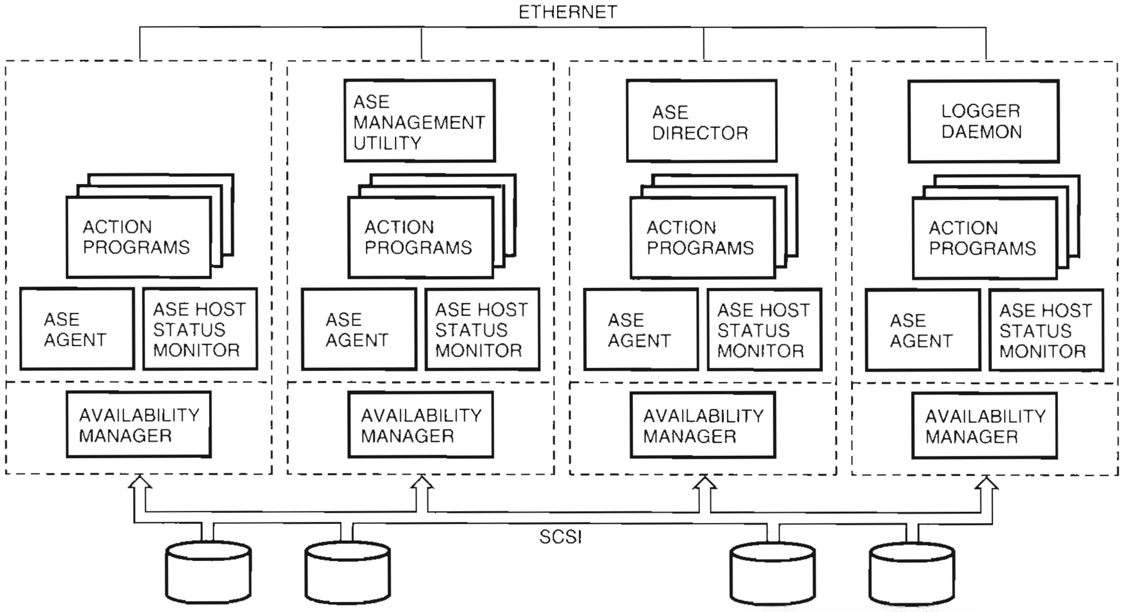


Figure 4
ASE Component Configuration

The ASE agent and the ASE director work as a team, reacting to component faults and performing failure recovery for services. The ASE events are generated by the ASE host status monitor and the availability manager (AM), a kernel subsystem. The ASE agents use the AM to detect device failures that pertain to ASE services. When a device failure is detected, the AM informs the ASE agent of the problem. The ASE agent then reports the problem to the ASE director if the failure results in service stoppage. For example, if the failed disk is part of an LSM mirrored set, the service is not affected by a single disk failure.

The ASE host status monitor sends host- or member-state change events to the ASE director. The ASE host status monitor uses both the networks and shared storage buses, SCSI buses, configured between the ASE members to determine the state of each member. This monitor uses the AM to provide periodic SCSI bus messaging through SCSI target-mode technology to hosts on the shared SCSI bus.

The ASE agent also uses the AM to provide device reservation control and device events. The ASE host status monitor repeatedly sends short messages, pings, to all other members and awaits a reply. If no reply is received within a prescribed time-out, the monitor moves to another interconnect until all paths have been exhausted without receiving a reply. If no reply on the shared network or any shared SCSI is received, the monitor presumes that the member is down and reports this to the ASE director. If any of the pings is successful and the member was previously down, the monitor reports that the member replying is up. If the only successful pings are SCSI-based, the ASE host status monitor reports that the members are experiencing a network partition. During a network partition, the ASE configuration and current service locations are frozen until the partition is resolved.

All ASE operations performed across the members use a common distributed logging facility. The logger daemon has the ability to generate multiple logs on each ASE member. The administrator uses the log to determine more detail about a particular service failover or configuration problem.

ASE Static and Dynamic States

As with most distributed applications, the ASE product must control and distribute state across a set of processes that can span several systems. This state takes two forms: static and dynamic. The static state is distributed in the ASE configuration database. This state is used to provide service availability configuration information and the ASE system membership list. Although most changes to the ASE configuration database are gathered through the ASE administrative command, all changes to the database are passed through a single point of control and distribution, the ASE director. The dynamic state includes changes in status of the base availability environment components and services. The state of a particular service, where and whether it is running, is also dynamic state that is held and controlled by the ASE director. Figure 5 depicts the flow of control through the ASE components.

ASE Director Creation

The ASE agents are responsible for controlling the placement and execution of the ASE director. Whenever an ASE member boots, it starts up the ASE agent to determine whether an ASE director needs to be started. This determination is based on whether an ASE director is already running on some member.

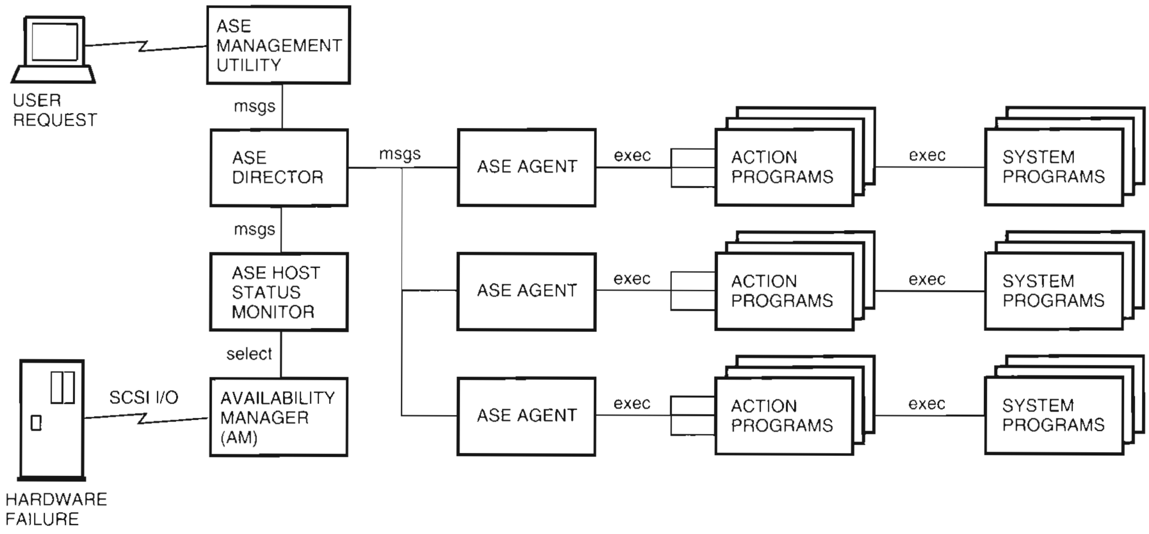


Figure 5
ASE Control Flow

If no ASE director is running and the ASE host status monitor is reporting that no other members are up, the ASE agent forks and executes the ASE director. Due to intermittent failures and the parallel initialization of members, an ASE configuration could find two ASE directors running on two different systems. As soon as the second director is discovered, the younger director is killed by the ASE agent on that system. The IP address of the primary network is used to determine which member should start a director when none is running.

ASE Director Design

The ASE director consists of four major components: the event manager, the strategist, the environment data manager, and the event controller. Figure 6 shows the relationship of the components of the ASE director.

The event manager component handles all incoming events and determines which subcomponent should service the event. The strategist component processes the event if it results in service relocation. The strategist creates an action plan to relocate the service. An action plan is a set of command lists designed to try all possible choices for processing the event. For example, if the event is to start a particular service, the generated plan orders the start attempts from the most desired member to the least desired member according to the service policy.

The environment data manager component is responsible for maintaining the current state of the ASE. The strategist will view the current state before creating an action plan. The event controller component oversees the execution of the action plan. Each of the command lists within the action plan is processed in parallel, whereas each command within a command list is processed serially. Functionally, this means that services can be started in parallel, and each service start-up can consist of a set of serially executed steps.

ASE Agent Design

The ASE agent is composed of the environment manager, the service manager, a second availability manager (AVMAN), and the configuration database manager. Figure 7 shows the ASE agent components.

All the ASE agent components use the message library as a common socket communications layer that allows the mixture of many outstanding requests and replies across several sockets. The environment manager component is responsible for the maintenance and initialization of the communications channels used by the ASE agent and the start-up of the ASE host status monitor and the ASE director. The environment manager is also responsible for handling all host-status events. For example, if the ASE host status monitor reports that the local node has lost connection to the network, the environment manager issues stop service actions on all services currently being served by the local node. This forced stop policy is based on the assumption that the services are being provided to clients on the network. A network that is down implies that no services are being provided; therefore, the service will be relocated to a member with healthy network connections.

If the ASE agent cannot make a connection to the ASE host status monitor during its initialization, the ASE host status monitor is started. The start-up of the ASE director is more complex because the ASE agent must ensure that only one ASE director is running in the ASE. This is accomplished by first obtaining the status of all the running ASE members. After the member status is commonly known across all ASE agents, the member with the highest IP address on the primary network is chosen to start up the ASE director. If two ASE directors are started, they must both make connections to all ASE agents in the ASE. In those rare cases when an ASE agent discovers two directors attempting to make connections, it will send

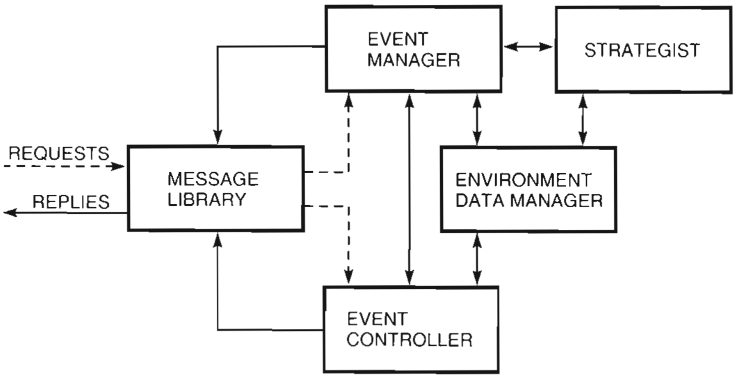


Figure 6
ASE Director

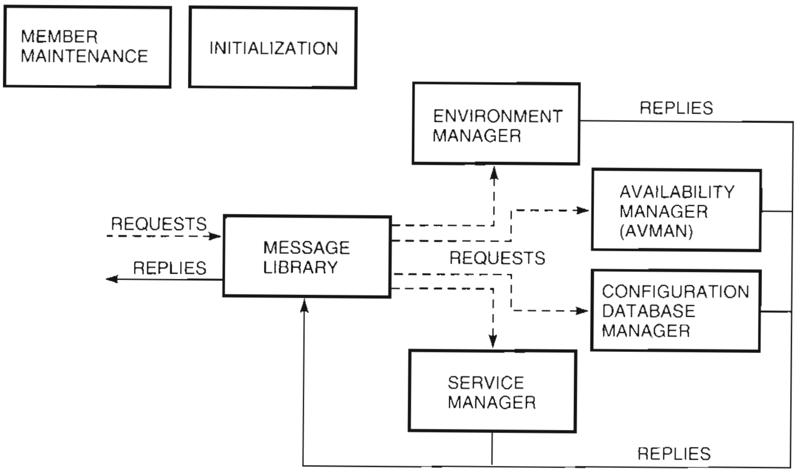


Figure 7
ASE Agent

an exit request to the younger director, the one with the newer start time.

The service manager component is responsible for performing operations on ASE services. The service manager performs operations that use specific service action programs or that determine and report status on services and their respective devices. The service actions are forked and executed as separate processes, children of the agent. This allows the ASE agent to continue handling other parallel actions or requests. The ASE agent is aware of only the general stop, start, add, delete, query, or check nature of the action. It is not aware of the specific application details required to implement these base availability functions. A more detailed description of the ASE service interfaces can be found in the section ASE Service Definition. When the service manager executes a stop or start service action that has device dependencies, the ASE agent provides the associated device reserves or unreserves to gain or release access to the device. Services and devices must be configured such that one device may be associated with only one service. A device may not belong to more than one service.

The agents' availability manager (AVMAN) component is called by the service manager to process a reserve or unreserve of a particular device for a service stop or start action. The AVMAN uses ioctl() calls to the AM to reserve the device, to invoke SCSI device pinging, and to register or unregister for the following AM events:

1. Device path failure—an I/O attempt failed on a reserved device due to a connectivity failure or bad device.
2. Device reservation failure—an I/O attempt failed on a reserve device because another node had reserved it.
3. Reservation reset—the SCSI reservation was lost on a particular device due to a bus reset.

A reservation reset occurs periodically as members reboot and join the ASE. The ASE agent reacts by rereserving the device and thereby continuing to provide the service. If the reservation reset persists, the ASE agent informs the ASE director. If a device path failure occurs, the ASE agent informs the ASE director of the device path failure so that another member can access the device and resume the service. The device reservation failure can occur only if another member has taken the reservation. This signifies to the ASE agent that an ASE director has decided to run this service on another member without first stopping it here.

The configuration database manager component handles requests that access the ASE configuration database. Working through the configuration database manager component, the ASE agent provides all access to the ASE configuration database for all other components of the ASE.

ASE Availability Manager Design

The availability manager (AM) is a kernel component of ASE that is responsible for providing SCSI device control and SCSI host pinging with target mode. The AM provides SCSI host pinging to the ASE host status monitor daemon through a set of ioctl() calls to the "/dev/am_host*" devices. As has been mentioned, the AM provides SCSI device control for pings and event notification to the ASE agent through ioctl() calls to the "/dev/ase" device. All ASE SCSI device controls for services and SCSI host pinging assume that all members are symmetrically configured with respect to SCSI storage bus addressing.

ASE Host Status Monitor Design

The ASE host status monitor (ASEHSM) component is responsible for sensing the status of members and interconnects used to communicate between members. As previously mentioned, this monitor is designed to provide periodic pinging of all network and SCSI interconnects that are symmetrically configured between ASE members. The ping rate is highest, 1 to 3 seconds per ping, on the first configured ASE network and SCSI bus. All other shared interconnects are pinged at a progressively slower rate to decrease the overhead while still providing some interconnectivity state. The ASE host status monitor provides member-state change events to both the ASE agent and the ASE director. The ASE agent initializes and updates the monitor when members are added or deleted from the ASE configuration database. The ASE host status monitor is designed to be flexible to new types of networks and storage buses as well as extensible to increased numbers of shared interconnects.

ASE Service Definition

ASE has provided an interface framework for available applications. This framework defines the availability configuration and failover processing stages to which an application must conform. The application interfaces consist of scripts that are used to start, stop, add, delete, query, check, and modify the particular service. Each script has the ability to order or stack a set of dependent scripts to suit a multilayered application. The NFS Service Failover section in this paper provides an example of a multilayered service that ASE supports "out of the box." ASE assumes that a service can be in one of the following states:

1. Nonexistent—not configured to run
2. Off-line—not to be run but configured to run
3. Unassigned—stopped and configured to run
4. Running—running on a member

At initialization, the ASE director presumes all configured services should be started except those in the off-line state. Whenever a new member joins the ASE, the add service action script is used to ensure that the new member has been configured to have the ability to run the service. The delete service script is used to remove the ability to run the service. The delete scripts are run whenever a service or member is deleted. The start service script is used to start the service on a particular member. The stop service is used to stop a service on a particular member. The check script is used to determine if a service is running on a particular member. The query script is used to determine if a particular device failure is sufficient to warrant failover.

ASE strives to keep a service in a known state. Consequently, if a start action script fails, ASE presumes that executing the stop action will return the service to an unassigned state. Likewise, if an add action fails, a delete action will return the service to a nonexistent state. If any action fails in the processing of an action list, the entire request has failed and is reported as such to the ASE director and in the log. For more details on ASE service action scripts, see the *Guide to the DECsafe Available Server*.[5]

NFS Service Failover

In this section, we present a walk-through of an NFS service failover. We presume that the reader is familiar with the workings of NFS.[1] The NFS service exports a file system that is remotely mounted by clients and locally mounted by the member that is providing the service. Other ASE members may also remotely mount the NFS file system to provide common access across all ASE members.

For this example, assume that we have set up an NFS service that is exporting a UNIX file system (UFS) named /foo_nfs. The UFS resides on an LSM disk group that is mirroring across two volumes that span four disks on two different SCSI buses. The NFS service is called foo_nfs and has been given its own IP address, 16.140.128.122. All remote clients who want to mount /foo_nfs will access the server using the service name foo_nfs and associated IP address 16.140.128.122. This network address information was distributed to the clients through the Berkeley Internet Name Domain (BIND) service or the network information service (NIS). If several NFS mount points are commonly used by all clients, they can be grouped into one service to reduce the number of IP addresses required. Although grouping directories exported from NFS into a single service reduces the management overhead, it also reduces flexibility for load balancing.

Further, assume that the NFS service foo_nfs has four clients. Two of the clients are the members of the ASE. The other two clients are non-Digital systems. For simplicity, the Sun and HP clients reside on the same network as the servers (but they need not). The ASE NFS service foo_nfs is currently running on the ASE member named MUNCH. The other ASE member is up and named ROLAIDS.

Enter our system administrator with his afternoon Big Gulp Soda. He places the Big Gulp Soda on top of MUNCH to free his hands for some serious console typing. Oh! We forgot one small aspect of the scenario. This ASE site is located in California. A small tremor later, and MUNCH gets a good taste of the Big Gulp Soda. Seconds later, MUNCH is very upset and fails. The ASE host status monitor on ROLAIDS stops receiving pings from MUNCH and declares MUNCH to be down. If the ASE director had been running on

MUNCH, then a new director is started on ROLAIDS to provide the much-needed relief. The ASE director now running on ROLAIDS determines that the foo_nfs service is not currently being served and issues a start plan for the service. The start action is passed to the local ASE agent since no other member is available. The ASE agent first reserves the disks associated with the foo_nfs service and runs the start action scripts. The start action scripts must begin by setting up LSM to address the mirrored disk group. The next action is to have UFS check and mount the /foo_nfs file system on the ASE hidden mount point /var/ase/mnt/foo_nfs. The hidden mount point helps to ensure that applications rarely access the mount point directly. This safeguard prevents an unmounting, which would stop the service. The next action scripts to be run are related to NFS. The NFS exports files must be adjusted to include the foo_nfs file system entry. This addition to the exports files is accomplished by adding and switching exports include files.

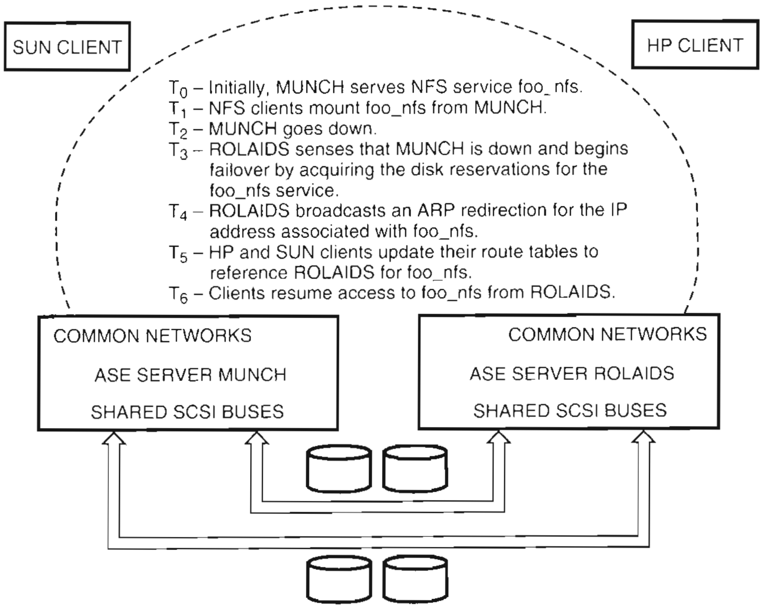
The action scripts then configure the service address (ifconfig alias command), which results in a broadcast of an Address Resolution Protocol (ARP) redirection packet to all listening clients to redirect their IP address mapping for 16.140.128.122 from MUNCH to ROLAIDS.[6] After all the ARP and router tables have been updated, the clients can resume communications with ROLAIDS for service foo_nfs. This entire process usually completes within ten seconds. The storage recovery process often contributes the longest duration. Figure 8 summarizes the time-sequenced events for an NFS service failover.

This scenario works because NFS is a stateless service. The server keeps no state on the clients, and the clients are willing to retry forever to regain access to their NFS service. Through proper mounting operations, all writes are done synchronously to disk such that a client will retry a write if it never receives a successful response.

If ASE is used to fail over a service that requires state, a mechanism has to be used to relocate the required state in order to start the service. The ASE product recommends that this state be written to file systems synchronously in periodic checkpoints. In this manner, the failover process could begin operation at the last successful checkpoint at the time the state disk area was mounted on the new system. If a more dynamic failover is required, the services must synchronize their state between members through some type of network transactions. This type of synchronization usually requires major changes to the design of the application.

Implementation and Development

We solved many interesting and logistically difficult issues during the development of the ASE product. Some of them have been discussed, such as the asymmetric versus symmetric SAD and distributed versus centralized policy. Others are mentioned in this section.



T_0 – Initially, MUNCH serves NFS service foo_nfs.
T_1 – NFS clients mount foo_nfs from MUNCH.
T_2 – MUNCH goes down.
T_3 – ROLAIDS senses that MUNCH is down and begins failover by acquiring the disk reservations for the foo_nfs service.
T_4 – ROLAIDS broadcasts an ARP redirection for the IP address associated with foo_nfs.
T_5 – HP and SUN clients update their route tables to reference ROLAIDS for foo_nfs.
T_6 – Clients resume access to foo_nfs from ROLAIDS.

Figure 8
Time-sequenced Events for NFS Failover

The SCSI Standard and High-availability Requirements

The SCSI standard provides two levels of requirements: mandatory and optional. The ASE requirements fall into the optional domain and are not normally implemented in SCSI controllers. In particular, ASE requires that two or more initiators (host SCSI controllers) coexist on the same SCSI bus. This feature allows for common access to shared storage. Normally, there is only one host per SCSI, so very little testing is done to ensure the electrical integrity of the bus when more than one host resides. Furthermore, to make the hosts uniquely addressable, we needed to assign SCSI IDs and not hardwire them. Lastly, to support its host-sensing needs, ASE requires that SCSI controllers respond to commands from another controller. This SCSI feature is called target-mode operation.

In addition to meeting the basic functional SCSI requirements, we had to deal with testing and qualification issues. When new or revised components were used in ways for which they were not originally tested, they could break; and invariably when a controller was first inserted into an ASE environment, we found problems. Additional qualifications were required for the SCSI cables, disks, and optional SCSI equipment. ASE required very specific hardware (and revisions of that hardware); it would be difficult to support off-the-shelf components.

Note, however, when all was said and done, only one piece of hardware, the Y cable, was invented for ASE. The Y cable allows the SCSI termination to be placed on the bus and not in the system. As a result, a system can be removed without corrupting the bus.

The challenge for the project was to convince the hardware groups within Digital that it was worth the expense of all the above requirements and yet provide cost-competitive controllers. Fortunately, we did; but these issues are ongoing in the development of new controllers and disks. Our investigation continues on alternatives to the target mode design. We also need to develop ways to reduce the qualification time and expense, while improving the overall quality and availability of the hardware.

NFS Modifications to Support High Availability

The issues and design of NFS failover could consume this entire paper. We discuss only the prominent points in this section.

NFS Client Notification

The first challenge we faced was to determine how to tell NFS clients which host was serving their files both during the initial mount and after a service relocation. The ideal solution would have been to provide an IP address that all nodes in the SAD could respond to. If clients knew only one address, all NFS packets would be sent to that address and we would never have to tell the client the location had changed. The main problem with this solution is performance. Each node in the SAD would receive all NFS traffic destined for all nodes. The system overhead for deciding whether to drop or keep the packet is very high. Also the more nodes and NFS services, the more likely it would be to saturate individual nodes. Unfortunately, this solution had to be rejected.

The next best solution, in our minds, is per service IP addresses. Each NFS service is assigned an IP address (not the real host address). Now each node in the SAD could respond to its own address and to the addresses assigned to the NFS services that it is running. The main issues with this approach are the following: (1) It could use many IP addresses and (2) It is more difficult to manage because of its many addresses. However, there were no performance trade-offs, and we could move services to locations in a way that was transparent to the NFS clients. Notifying the clients after a relocation turned out to be easy because of a standard feature in the ARP that we could access through the ifconfig alias command of the Digital UNIX operating system.[6] Essentially, all clients have a cache of translations for IP addresses to physical addresses. The ARP feature, which we referred to as ARP redirection, allows us to invalidate a client-cached entry and replace it with a new one. The ifconfig command indirectly generates an ARP redirection message. As a result, the client NFS software believes it is sending to the same address, but the network layer is sending it to a different node.

Similar functionality could have been achieved by requiring multiple network controllers connected to a single network wire on the SAD nodes. This solution, however, requires more expense in hardware and is less flexible since there is only one address per board. Essentially, the latter means the granularity of NFS services would be much larger and could not be distributed among many SAD nodes without a great deal of hardware.

NFS Duplicate Request Cache

The NFS duplicate request cache improves the performance and correctness of an NFS server.[7] Although the duplicate request cache is not preserved across relocations, we did not view this as a significant problem because this cache is not preserved across reboots.

Other Modifications: Lock Daemons and mountd

We modified only two pieces of software related to NFS failover: the lock daemon and the mountd. We wanted the lock daemon to distinguish the locks associated with a specific service address so that only those locks would be removed during a relocation. After the service is relocated, we rely on the existing lock reestablishment protocol. We modified the mountd to support NFS loopback mounting on the SAD, so that a file system could be accessed directly on the SAD (as opposed to a remote client) and yet be relocated transparently.

Future of ASE

Digital's ASE product was designed to address a small, symmetrically configured availability domain. The implementation of the ASE product was constrained by time, resources, and impact or change in the base system. Consequently, the ASE product lacks extensibility to larger asymmetric configurations and to more complex application availability requirements, e.g., support of concurrent distributed applications. The next-generation availability product must be designed to be extensible to varying hardware configurations and to be flexible to various application availability requirements.

Acknowledgments

We thank the following people for contributing to this document through their consultation and artwork: Terry Linsey, Mark Longo, Sue Ries, Hai Huang, and Wayne Cardoza.

References

1. Sun Microsystems, Inc., *NFS Network File System Protocol Specification,* RFC 1094 (Network Information Center, SRI International, 1989).

2. J. Gray, "High Availability Computer Systems," *IEEE Computer* (September 1991).

3. A. Bhide, S. Morgan, and E. Elnozahy, "A Highly Available Network File Server," *Conference Proceedings from the Usenix Conference,* Dallas, Tex. (Winter 1991).

4. W. Cardoza, F. Glover, and W. Snaman, "Design of the Digital UNIX Cluster System," *Digital Technical Journal,* forthcoming 1996.

5. *Guide to the DECsafe Available Server* (Maynard, Mass.: Digital Equipment Corporation, 1995).

6. D. Plummer, *Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48-bit Ethernet Address for Transmission on Ethernet Hardware,* RFC 0826 (Network Information Center, SRI International, 1982).

7. C. Juszczak, "Improving the Performance and Correctness of an NFS Server," *Conference Proceedings from the Usenix Conference,* San Diego, Calif. (Winter 1989).

Biographies

Lawrence S. Cohen
Larry Cohen led the Available Server Environment project. He is a principal software engineer in Digital's UNIX Engineering Group, where he is currently working on Digital's UNIX cluster products. Since joining Digital in 1983, he has written network and terminal device drivers and worked on the original ULTRIX port of BSD sockets and the TCP/IP implementation from BSD UNIX. Larry also participated in the implementation of Digital's I/O port architecture on ULTRIX and in the port of NFS version 2.0 to the DEC OSF/1 version of UNIX. Larry was previously employed at Bell Labs, where he worked on the UNIX to UNIX Copy Program (UUCP). He has a B.S. in math (1976) and an M.S. in computer science (1981), both from the University of Connecticut.

John H. Williams
John Williams is a principal software engineer in Digital's UNIX Engineering Group. John led the advanced development efforts for the UNIX cluster product and was the project leader for the DECsafe Available Server Environment version 1.1 and version 1.2. Before that, John designed and implemented the security interface architecture for the DEC OSF/1 operating system. Currently, John is responsible for the UNIX cluster management features. John received a B.S. in computer science from the Michigan Technological University in 1978.

Michael Palmer
Jeffrey M. Russo

Parasight: Debugging and Analyzing Real-time Applications under Digital UNIX

Conventional UNIX debug and analysis tools, with their static debugging model and low-resolution-sampling profiling techniques, are not effective in dealing with real-time applications. Encore Computer Corporation has developed Parasight, a set of debug and analysis tools for real-time applications. The Parasight tool set can debug running programs, debug multiple programs, constantly monitor local and global variables, and perform on-the-fly execution analysis. Thus, Parasight provides much improved debug and analysis capabilities, which application developers can use on both static and dynamic applications. Parasight can be used on any of Digital's Alpha platforms running under the Digital UNIX operating system.

Because of their time-critical nature, real-time applications do not respond well to the perturbations that conventional UNIX debug and analysis tools cause. For instance, the static debugging model of the dbx debugger requires that a program be stopped before it can be debugged. Also, execution analysis using the profiling techniques of the prof profiler often provide erroneous results for real-time applications because of the low-resolution sampling employed.

This paper describes the critical aspects of debugging real-time applications, the deficiencies found in conventional UNIX tools, and the methodology Encore Computer Corporation used to develop Parasight, a set of easy-to-use graphical user interface tools that debug and perform execution analysis on real-time programs while they are running. Parasight can be used on any of Digital's Alpha platforms that operate under the Digital UNIX operating system.

Real-time Applications

Real-time applications perform a wide variety of functions, from flying state-of-the-art military aircraft to controlling nuclear power plants. All real-time applications have one common denominator: They must complete their calculations before a deadline expires. Taking too long to calculate the correct answer can have just as detrimental an effect as arriving at an incorrect answer; either result could cause an aircraft to crash or a nuclear power plant to experience a meltdown.

Most real-time applications consist of one or more programs that are scheduled to run in response to an event. The triggering event is usually transmitted in the form of an interrupt and can be generated randomly by an external event or regularly by a interval timer running at a fixed rate, such as 60 times per second. Once the interrupt is received, the application must perform the allotted task before the next interrupt occurs.

The elements of a real-time application communicate with each other dynamically; that is, the results of the calculations of one element are used immediately for the calculations of another element. Real-time applications are often referred to as dynamic applications, since they react dynamically to changes in their environment and often refer to elapsed time in their calculations. In contrast, static applications have results that rarely depend on changes in their environment or on elapsed time.

The Problems Associated with Debugging and Analyzing Real-time Applications Using Conventional UNIX Tools

Debugging a real-time application during execution, debugging and analyzing multiple programs, constantly monitoring variables, and analyzing program execution are all activities that debug and analysis tools have to deal with. This section discusses the capabilities and limitations of conventional UNIX tools and describes the features required of effective real-time debug and analysis tools.

Running Programs

Debugging a static program typically involves controlling the execution flow and examining the values of variables within the program. Stopping a real-time program or even delaying it by single stepping, however, is usually not possible without adversely affecting the application. Debugging real-time applications is therefore limited to examining the values of program variables while the program is still running.

Conventional UNIX debuggers are not able to examine variables during program execution and therefore cannot be employed on running real-time applications. Consequently, these debuggers are useful only in the early stages of real-time program development, essentially while the program is still static.

The traditional methods of debugging real-time applications involve placing all the critical data into one or more global, shared memory regions. A data-monitoring tool, usually written by the user, runs as a normal UNIX process and attaches to the global region. The tool can then be used to inspect and/or change the values of the global variables. This technique is nonintrusive in that it does not affect the real-time application programs in any way. Unfortunately, the debugging is restricted to global data, and, unless the programs are designed with this in mind, this restriction can be a severe limitation. Modifying existing programs to change local data into global data for debugging purposes can result in a whole new set of problems in managing the separation of data.

An effective real-time debugging tool must be able to attach to a running program without stopping it and then be able to nonintrusively inspect and/or change the global data.

Debugging and Analyzing Multiple Programs

Real-time applications typically consist of several programs working together. Invoking multiple copies of the dbx debugger to debug each program individually is cumbersome and precludes studying the interaction between programs.

A real-time debugger must be able to work with one or more programs at the same time, providing the user with an integrated and cohesive debugging environment.

Monitoring Variables

The one-shot variable evaluation capability of conventional UNIX debuggers is of limited use for programs that are running. These debugging tools provide the user with only one previous value of a variable, not necessarily the current value.

A real-time debugger must be able to constantly monitor the values of global variables. The minimum and maximum values that each variable attained should optionally be available as a record of transient conditions.

Execution Analysis (Profiling)

Since performance is important in real-time applications, program execution analysis is often needed to locate areas of a program where the performance could be improved. A real-time application may have a strict execution order requirement, whereby one routine must run prior to the execution of another routine. This requirement may be accomplished easily if the routines are in the same program; however, often the routines are in different programs or are executing on different CPUs in a symmetric multiprocessing (SMP) environment.

The Digital UNIX profiling tools provide two kinds of execution analysis:

1. PC sampling, which involves interrupting the program periodically to record the value of the program counter.
2. Block counting, which inserts profiling code at key points in the program to count the number of times each basic block of code executes. (A basic block is a region of the program that can be entered only at the beginning and exited only at the end.)

Both techniques involve the following basic steps:

1. Preprocess the program to produce the desired profiling information.
2. Execute the program to produce a profiling data file.
3. Postprocess the program with the profiling tools to view the data collected.

The normal sampling period employed by the PC-sampling method is based on the hard clock (CLOCK_REALTIME) of the Digital UNIX operating system. This method results in 1,024 samples being taken per second, which provides a timing resolution of 976 microseconds, or approximately 1 millisecond.

The routines that make up a real-time application typically take from a few microseconds to several milliseconds to execute. Attempting to measure the execution time of routines that take less than 1 millisecond to execute with a clock resolution of 1 millisecond can lead to erroneous results. A test on a 150-megahertz (MHz) Alpha 21064 CPU showed that the prof tool, using the normal PC-sampling rate, reported the execution time of a routine to be 4 milliseconds when the true execution time was 20 microseconds. (The true execution time was measured using the Parasight tool set.)

It is possible to increase the sampling rate using the uprofile utility, but doing so also proportionally increases the number of interrupts per second that the system must handle. For instance, to obtain even 10-microsecond resolution would require the system to handle 100,000 interrupts per second. This amount of interrupt activity would rapidly swamp the system, leaving little or no CPU time to execute the program being instrumented. The PC-sampling method of execution analysis is therefore not suitable for the short execution times typical in real-time application routines.

The block-counting method, although capable of high-resolution measurement, suffers from the inability of the pixie utility to work with programs that receive signals. Most real-time applications use signals for program scheduling and are therefore disqualified from using the block-counting method.

In addition to the problems just discussed, the traditional UNIX profiling tools are unsuitable for real-time program execution analysis for the following reasons:

- A program must be preprocessed for profiling prior to execution. Adding or removing profiling requires stopping, processing, and restarting the program. This assumes that the problem area is known before the application starts to run. If a problem suddenly develops after an uninstrumented program has been running for 24 hours, the user will have lost the opportunity to determine which routine is causing the problem.
- A program must be profiled as a whole, unless source code modifications are made to the program to control the profiling. This can cause excessive overhead, which real-time programs usually cannot tolerate.
- The profiling results cannot be seen until the program terminates, unless source code modifications are made to the program to permit the results to be dumped on command. The user needs to see the results while the program is running and often needs to repeat a test several times to get the desired results. Stopping and restarting the application once for each test could be laborious.
- Only the average and cumulative times for each routine are available. That is, the individual execution times for each call to a routine are not available. This also precludes the examination of the calling sequence.
- The results cannot be cross-correlated between programs to provide information about the relative calling sequences between programs or across processors.

A real-time execution analysis tool must operate with sufficient resolution to measure the execution time of a routine that may take 10 microseconds to execute. The instrumentation should be dynamically insertable into the current areas of interest and should be able to move to new areas of interest as required—all without stopping and restarting the real-time application.

Parasight: A Solution for Real-time Debugging and Program Analysis

Parasight is an integrated set of real-time debugging and analysis tools with a graphical user interface. The tool set consists of a debugger (Debug), a data monitor (DataMon), and a program analysis tool (Paragraph). The Parasight tool set solves the real-time deficiencies found in dbx, prof, and the other conventional UNIX debug and analysis tools used under the Digital UNIX operating system. Parasight is able to debug applications in either a dynamic (running) or a static (stopped) state; it can perform debugging and program execution analysis on several programs simultaneously, without adversely affecting the dynamics of time-critical applications.

Parasight's Foundation

The Parasight tool set features the use of a symbol table, the /prof file system, global memory, and scanpoints.

The Symbol Table, .pg File, and /proc File System Parasight's source of knowledge about the target application is derived from the symbol table and the .pg file. Both are generated at compile time as a result of the -para special compiler option.

Parasight manipulates target applications by using the /proc file system services available under the Digital UNIX operating system. The /proc file system enables Parasight to control the program flow and to read and write any memory address in the target application.

Global Data Just as the traditional means of debugging real-time applications depends on global memory regions, Parasight uses the global memory access concept as the basis for accomplishing most of its advanced capabilities. Parasight either accesses the target program data directly, through the use of /proc, or uses global memory to access data gathered for Parasight by one of its scanpoints.

Scanpoints The Parasight tool set uses global memory access whenever possible to provide nonintrusive access to the target application. Certain functions, however, require access to data that is local to a program. Parasight accesses this data through small segments of code called scanpoints.

A scanpoint is a function that is dynamically inserted into the target program by Parasight. The scanpoint function then runs in the same context as the target program and thus has access to all the local data of the program. The scanpoint function works as an agent for Parasight, gathering data that Parasight does not have direct access to. The Parasight tool set uses two principal types of scanpoints: datamon-scanpoints, which are used by DataMon to perform local data monitoring, and sensor-scanpoints, which are used by Paragraph to perform program execution analysis.

Inserting the scanpoints does not require modifying the application's source code or preprocessing the application's object code. The only requirement is to link each program with the special -para option. This adds a memory buffer to the target program for use by Parasight. The buffer is benign until used by Parasight.

Parasight dynamically inserts scanpoints by using the /proc service to build a scanpoint template in the special buffer of the target program. This can occur even while the program is running. The template code contains the functionality to

- Save the register state that existed when the program counter was at the scanpoint insertion location
- Set up the arguments to the scanpoint function, including the register state
- Call the scanpoint function
- Restore the register state
- Execute the instruction that was originally at the insertion location
- Branch back to the instruction following the insertion location

Parasight then dynamically alters the template code according to the insertion location and the instruction contained therein. If the instruction was a branch control instruction, Parasight alters the instruction's displacement so that the location corresponds to the instruction's new displaced location within the template. All other instructions, including jump control instructions, do not require altering and are simply copied to the new displaced location.

Once this code is constructed in the buffer, Parasight completes the scanpoint insertion process by overwriting the instruction at the insertion location with a branch to the newly generated scanpoint template. The fixed instruction length of Digital's Alpha microprocessors simplifies this step enormously.

It is important to note that the scanpoint is built by Parasight, not the target program. The target program is affected only by the final step of the scanpoint insertion, when Parasight overwrites the instruction at the insertion location. This design prevents excessive interference of the target program. Scanpoints are written in highly optimized code to minimize the impact on the target application when they are executed.

Parasight dynamically deletes scanpoints by writing back the original instruction at the insertion location. This design allows Parasight to disable a scanpoint even if the scanpoint function has not completed.

Meeting Requirements

Parasight has the capabilities required of effective real-time debugging and analysis tools.

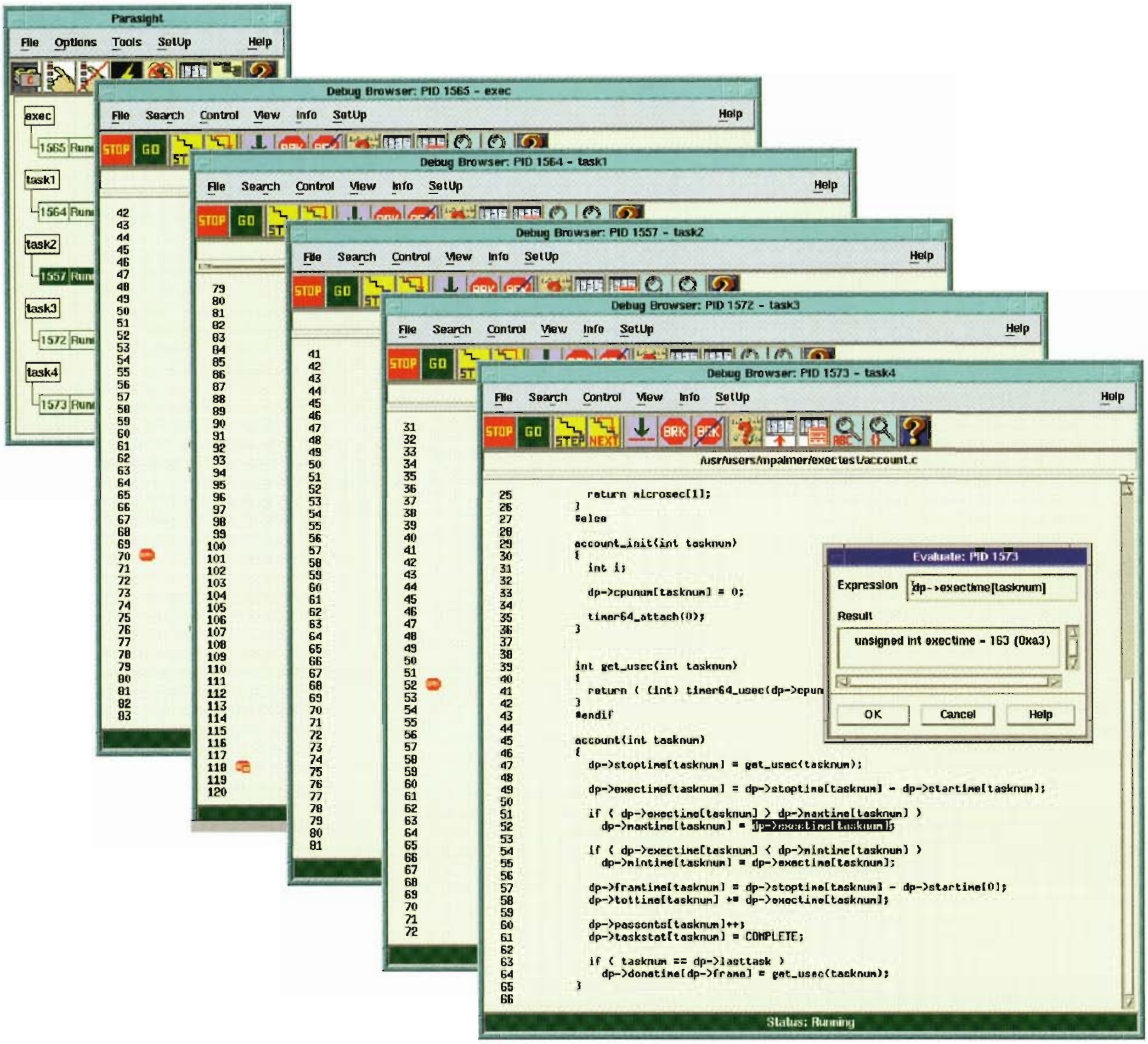
Debugging Running Programs Conventional UNIX debuggers deliberately stop a program when attaching to it, because these tools do not operate on running programs. When Parasight's debugger, Debug, attaches to a program, there is no impact on the program.

Conventional UNIX debuggers refuse to access any data while a program is running, even though global data resides at fixed memory locations that are accessible at all times through the /proc service. The reason for this limitation of the conventional UNIX tools is unclear. Parasight's debugger is able to examine and to change the value of any global data while the program is running or stopped.

Conventional UNIX tools also refuse to set any breakpoints in a program while the program is running. Again, the reason for this constraint is unknown. Parasight's debugger is able to insert breakpoints into running programs, a feature that is valuable in debugging error conditions in real-time applications.

Debugging Multiple Programs Parasight's Debug, DataMon, and Paragraph components constitute an integrated set of tools capable of working on one or more applications simultaneously, as shown in Figure 1. The Parasight main window displays the programs (and any children they create) attached to Parasight. The window also provides an easy mechanism to access the Parasight tool for each specific program.

Monitoring Variables Constantly Parasight's DataMon tool allows the user to simultaneously monitor the values of any local or global variables in one or more stopped or running programs. Parasight constantly monitors the values and shows any change on the DataMon display screen. DataMon is also capable of displaying the minimum, maximum, and average


Parasight
File Options Tools SetUp Help
exec
1565 Run
task1
1564 Run
task2
1557 Run
task3
1572 Run
task4
1573 Run
Debug Browser: PID 1565 - exec
Debug Browser: PID 1564 - task1
Debug Browser: PID 1557 - task2
Debug Browser: PID 1572 - task3
Debug Browser: PID 1573 - task4
File Search Control View Info SetUp Help
STOP GO STEP NEXT BRK BRK ABC
/usr/users/mpalmer/exectest/account.c
Status: Running

```
            return microsec[1];
        }
        #else

        account_init(int tasknum)
        {
          int i;

          dp->cpunum[tasknum] = 0;

          timer64_attach(0);
        }


        int get_usec(int tasknum)
        {
          return ( (int) timer64_usec(dp->cpun
        }
        #endif

        account(int tasknum)
        {
          dp->stoptime[tasknum] = get_usec(tasknum);

          dp->exectime[tasknum] = dp->stoptime[tasknum] - dp->startime[tasknum];

          if ( dp->exectime[tasknum] > dp->maxtime[tasknum] )
            dp->maxtime[tasknum] = dp->exectime[tasknum];

          if ( dp->exectime[tasknum] < dp->mintime[tasknum] )
            dp->mintime[tasknum] = dp->exectime[tasknum];

          dp->frametime[tasknum] = dp->stoptime[tasknum] - dp->startime[0];
          dp->tottime[tasknum] += dp->exectime[tasknum];

          dp->passcnts[tasknum]++;
          dp->taskstat[tasknum] = COMPLETE;

          if ( tasknum == dp->lasttask )
            dp->donetime[dp->frame] = get_usec(tasknum);
        }
```

Evaluate: PID 1573

Expression: dp->exectime[tasknum]

Result: unsigned int exectime = 163 (0xa3)

OK Cancel Help

Figure 1
Parasight's Debugger Working with Five Tasks Simultaneously

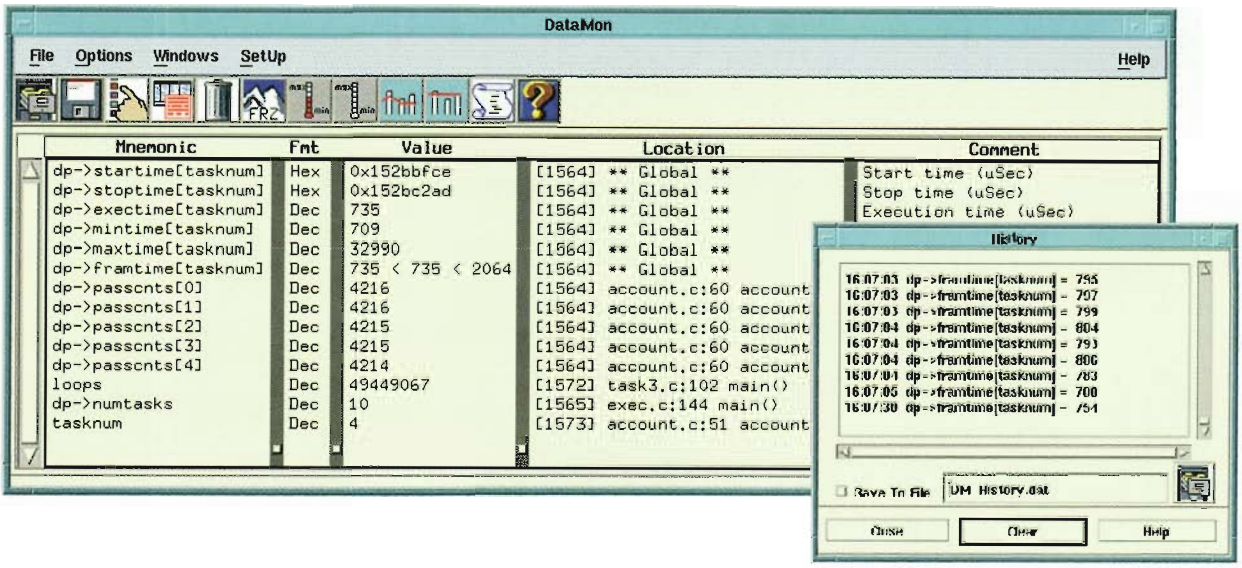
values attained for any variable. A scrolling history display along with a time stamp is also available for solving transient problems.

The variables to be monitored can be selected using the mouse on the Debug browser or entered into a dialog box using the keyboard. The DataMon graphical user interface has a point-and-edit capability, which allows the user to edit the mnemonic data (i.e., name, display format, value, location, or comment) directly on the screen. The user can store mnemonic lists on disk for fast retrieval when required. Figure 2 shows a DataMon display screen.

DataMon is able to monitor global data completely and nonintrusively using the /proc service and uses a datamon-scanpoint to implement local data monitoring. The datamon-scanpoint is attached to the DataMon database, which is a shared memory region connecting all the scanpoints and the DataMon display program. The datamon-scanpoints deposit the values of local data into the database for the display program to show on the screen. Datamon-scanpoints are also used to change the values of local data, depositing the value from the database into the specified variable.

DataMon uses the Debug tool's expression evaluator to parse the required mnemonic to derive the location of the value to be displayed. This may include register access for local variables saved on the stack. Multiple mnemonics can be monitored locally at the same location since a datamon-scanpoint function can traverse a list of mnemonics to be monitored.

Note that DataMon monitors data asynchronously; therefore, DataMon cannot guarantee to display every



Mnemonic	Fmt	Value	Location	Comment
dp->startime[tasknum]	Hex	0x152bbfce	[1564] ** Global **	Start time (uSec)
dp->stoptime[tasknum]	Hex	0x152bc2ad	[1564] ** Global **	Stop time (uSec)
dp->exectime[tasknum]	Dec	735	[1564] ** Global **	Execution time (uSec)
dp->mintime[tasknum]	Dec	709	[1564] ** Global **	
dp->maxtime[tasknum]	Dec	32990	[1564] ** Global **	
dp->framtime[tasknum]	Dec	735 < 735 < 2064	[1564] ** Global **	
dp->passcnts[0]	Dec	4216	[1564] account.c:60 account	
dp->passcnts[1]	Dec	4216	[1564] account.c:60 account	
dp->passcnts[2]	Dec	4215	[1564] account.c:60 account	
dp->passcnts[3]	Dec	4215	[1564] account.c:60 account	
dp->passcnts[4]	Dec	4214	[1564] account.c:60 account	
loops	Dec	49449067	[1572] task3.c:102 main()	
dp->numtasks	Dec	10	[1565] exec.c:144 main()	
tasknum	Dec	4	[1573] account.c:51 account	

```
16:07:03  dp->framtime[tasknum] =  795
16:07:03  dp->framtime[tasknum] -  797
16:07:03  dp->framtime[tasknum] =  799
16:07:04  dp->framtime[tasknum] -  804
16:07:04  dp->framtime[tasknum] =  793
16:07:04  dp->framtime[tasknum] -  806
16:07:04  dp->framtime[tasknum] -  783
16:07:05  dp->framtime[tasknum] =  700
16:07:30  dp->framtime[tasknum] -  754
```

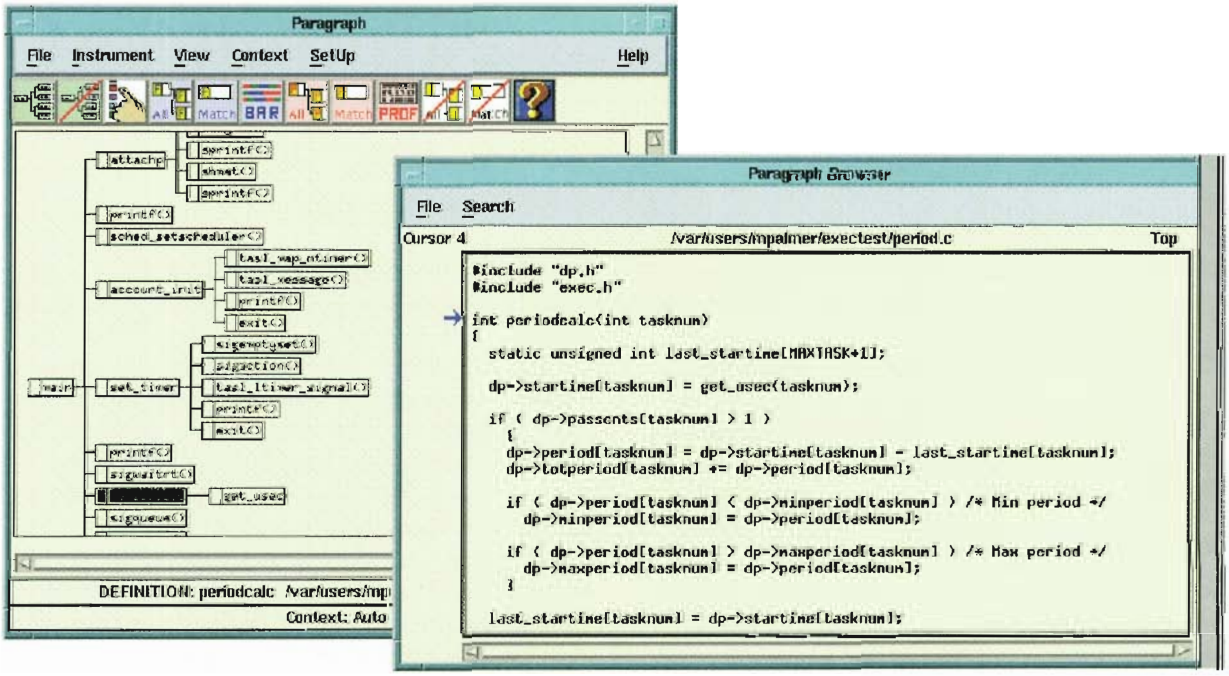
□ Save To File UM History.dat

Close Clear Help

Figure 2
The DataMon Display Screen with History Window

value that the variables reach. For global data, Parasight records only the minimum and maximum values that DataMon sees. For local data, however, the scanpoint keeps track of the minimum, maximum, and average values, so these can be guaranteed. Parasight can also monitor global data by using a datamon-scanpoint to monitor the value at a particular line of code.

On-the-Fly Execution Analysis Paragraph displays static source-code call graphs of the application's programs, illustrating the hierarchy of function calls, system calls, and statement-level control flow. Point-and-click operations allow the user to quickly view the source code for any program or function, thus simplifying the task of analyzing source code. Figure 3 shows a Paragraph call graph and browser.



DEFINITION: periodcalc /var/users/mp

Context: Auto

/var/users/mpalmer/exectest/period.c

```
#include "dp.h"
#include "exec.h"

int periodcalc(int tasknum)
{
  static unsigned int last_startime[MAXTASK+1];

  dp->startime[tasknum] = get_usec(tasknum);

  if ( dp->passcnts[tasknum] > 1 )
    {
    dp->period[tasknum] = dp->startime[tasknum] - last_startime[tasknum];
    dp->totperiod[tasknum] += dp->period[tasknum];

    if ( dp->period[tasknum] < dp->minperiod[tasknum] ) /* Min period */
      dp->minperiod[tasknum] = dp->period[tasknum];

    if ( dp->period[tasknum] > dp->maxperiod[tasknum] ) /* Max period */
      dp->maxperiod[tasknum] = dp->period[tasknum];
    }

  last_startime[tasknum] = dp->startime[tasknum];
```

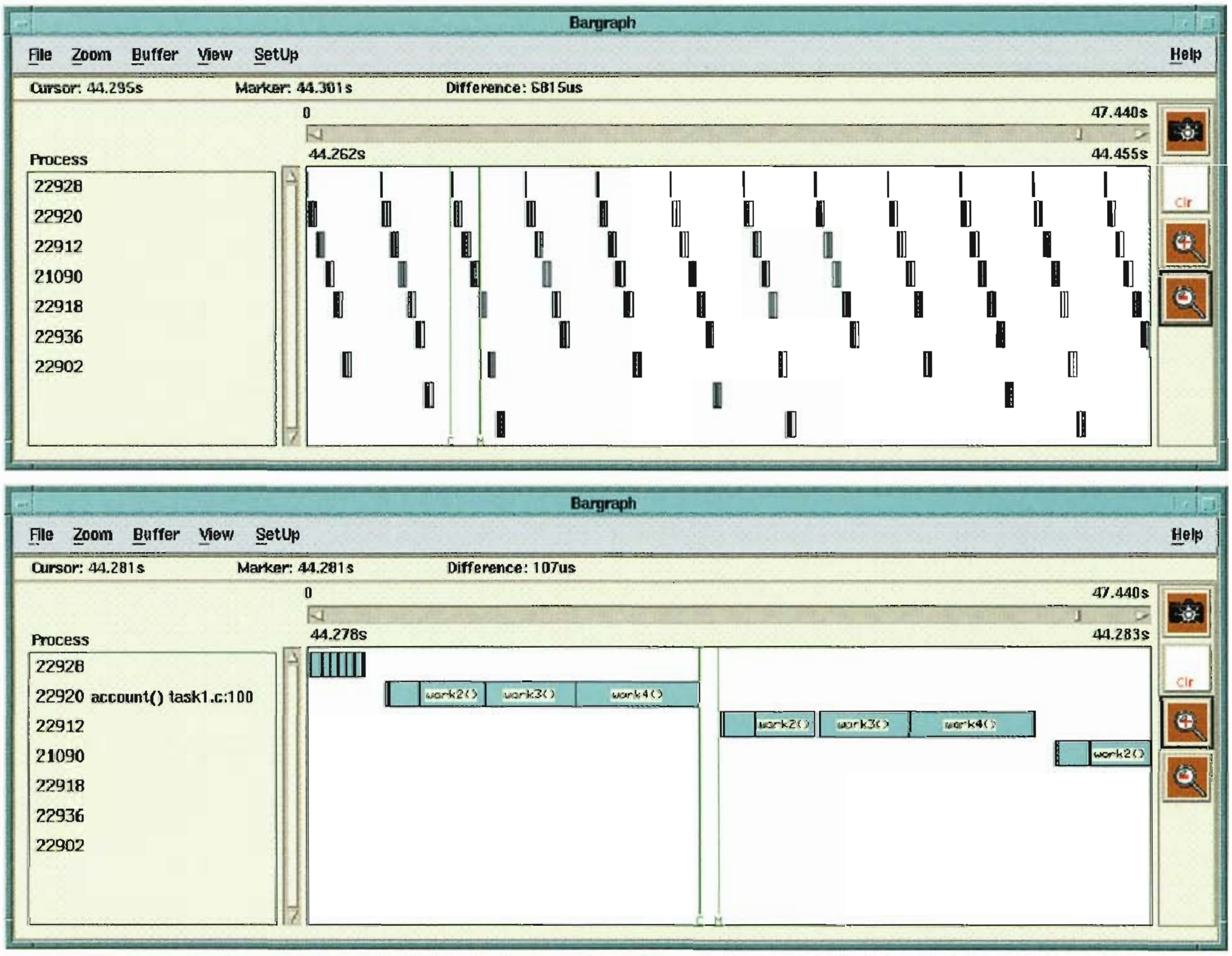
Figure 3
Paragraph Call Graph and Browser

Call graphs are also used to define where to insert instrumentation in an application. The instrumentation is used to perform execution timing analysis on a part or the whole of one or more of an application's programs. The instrumentation is inserted dynamically into a running program, without the need for source-level changes or object code preprocessing and without significantly affecting the dynamics of a running application. The inserted instrumentation may be deleted or added to at any time.

Paragraph uses sensor-scanpoints to measure how long a function takes to execute. The sensor-scanpoint function is placed at a branch-to-subroutine instruction. The function takes a time stamp from a nanosecond-resolution timer before and after the instruction to note the exact time the function started and ended. The sensor-scanpoints are attached to the Paragraph database, a shared region accessible to the sensor-scanpoints and Paragraph. Data is written into the database each time an instrumented function is executed. The results of the instrumentation may be viewed immediately, even while the program is running. The graphical view shows each function call as it occurred in time. Each program has a different bar, so the user can determine the relative time between functions called in different programs or even across multiple processors in an SMP environment. The zoom capability may be used to measure time periods down to a single microsecond. Figure 4 shows the Paragraph graphical display, called Bargraph, and the zoom capability.

Data gathering is continuous until the instrumentation is removed, so new data can be added onto the previous snapshot's view at any time. Multiple Bargraph windows can be used to recall previously saved timing data to easily compare current results with past results.

The nanosecond-resolution timer used by Paragraph is derived from the process control counter (PCC) register available on all Alpha microprocessors. This 32-bit, free-running timer operates at the same rate as the microprocessor and therefore provides a



Cursor: 44.295s Marker: 44.301s Difference: 6815us

0 47.440s

44.262s 44.455s

Process
22928
22920
22912
21090
22918
22936
22902

Cursor: 44.281s Marker: 44.281s Difference: 107us

0 47.440s

44.278s 44.283s

Process
22928
22920 account() task1.c:100
22912
21090
22918
22936
22902

Figure 4
The Paragraph Graphical Display, Bargraph, Showing Zoom Capability

3.6-nanosecond-resolution timer on a 275-MHz Alpha CPU. Unfortunately, since it is only a 32-bit timer, it wraps every 15.6 seconds. Parasight keeps track of the wrap count to create a 64-bit timer that allows problem-free timing for more than 2,000 years!

Adverse Effects

Although, ideally, the Parasight tool set should be completely nonintrusive and thus not affect the application in any way, such operation is not completely achievable for all functions. Capabilities such as inspecting (Debug) and monitoring (DataMon) global variables require no intrusion to implement; however, monitoring local variables and analyzing program execution do require a small amount of intrusion.

While most real-time applications cannot tolerate exceeding the time available for the completion of the task, they do have some spare time available after completing the task. Without this spare time, the risk of exceeding the deadline before program completion would be too great. This spare time can be used judiciously for the mildly intrusive functions of Parasight.

Summary

This paper discusses several capabilities required to effectively debug and analyze real-time applications. These capabilities include debugging of running programs, constant monitoring of variables, and on-the-fly execution analysis. The paper also details some of the problems associated with conventional UNIX tools, such as the inability to debug running programs, the adverse effects on target programs, the erroneous profiling results, and the cumbersome operation. Encore Computer Corporation's Parasight tool set offers a solution to these difficult problems. The paper describes the methodology behind the product and the capabilities that make Parasight an invaluable tool for debugging and analyzing real-time applications.

Acknowledgments

The authors would like to acknowledge the efforts of the following Parasight team members for their contributions to the product: Raghuveer Chakravarthi, Dileep Katta, Carlos Gonzalez, Deborah Grimstead, and Ken Shaffer.

General References

Z. Aral, I. Gertner, and G. Shaffer, "Efficient Debugging Primitives for Multiprocessors" (Fort Lauderdale, Fla.: Encore Computer Corporation, 1989).

DEC OSF/1 Programmer's Guide, Section 6 (Maynard, Mass.: Digital Equipment Corporation, August 1994).

Biographies

Michael Palmer

Michael Palmer is a principal member of Encore Computer Corporation's technical staff and has led the Parasight team for the past three years. Prior to joining Encore in 1991, Mike worked for several major flight simulation vendors throughout the world, advancing from computer systems engineer to lead software engineer for a $50 million, dual-dome tactical fighter simulator. He has used his real-time simulation background to mold Parasight into a leading tool set for real-time development. Mike holds a B.Sc. (Honors) in electronics from Newcastle Polytechnic, Newcastle upon Tyne, England.

Jeffrey M. Russo

Jeff Russo has been employed by IBM since June 1995. He is an Advisory Programmer working as a team leader for the OS/2 operating system. Prior to joining IBM, Jeff worked at Encore Computer Corporation for 10 years, advancing from the position of software engineer to that of Senior Section Manager responsible for several real-time software groups. He has significant experience with real-time, microkernel-based operating systems, as well as with the accompanying critical, real-time tool set. Jeff earned a B.S. in computer engineering from the University of Florida in 1985.

Call for Authors from Digital Press

Digital Press is an imprint of Butterworth-Heinemann, a major international publisher of professional books and a member of the Reed Elsevier group. Digital Press is the authorized publisher for Digital Equipment Corporation: The two companies are working in partnership to identify and publish new books under the Digital Press imprint and create opportunities for authors to publish their work.

Digital Press is committed to publishing high-quality books on a wide variety of subjects. We would like to hear from you if you are writing or thinking about writing a book.

Contact: Mike Cash, Digital Press Manager, or
Liz McCarthy, Assistant Editor

DIGITAL PRESS
313 Washington Street
Newton, MA 02158-1626
U.S.A.
Tel: (617) 928-2649, Fax: (617) 928-2640
E-mail: Mike.Cash@BHein.rel.co.uk or
LizMc@world.std.com

digital™

File
Options
digital
Capture rate (frames/sec): 25.5
.1 0.2 0.3 0.4 0.5 0.6
1.8348 Freq: 559.86 Value: 65

ISSN 0898-901X