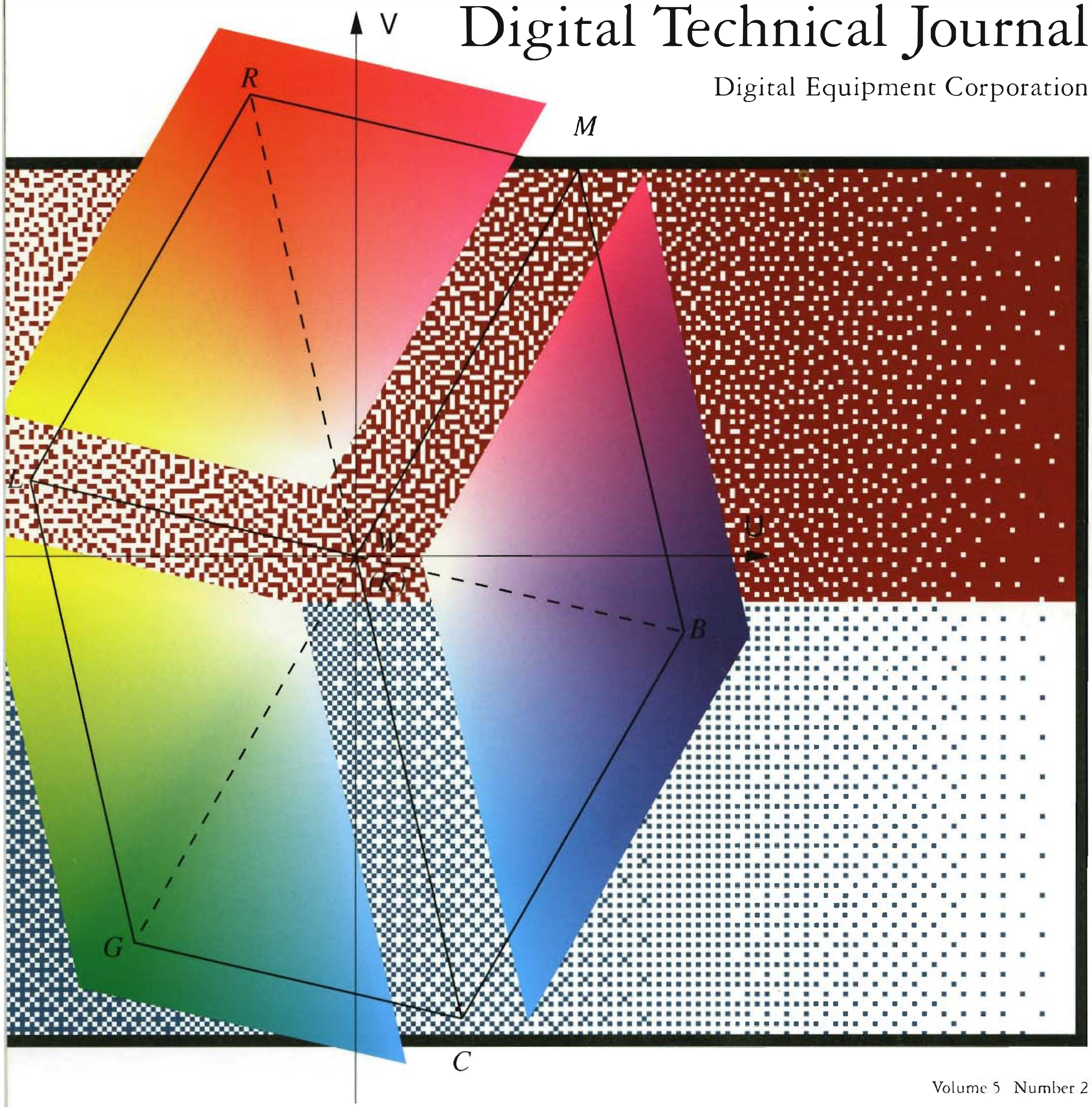
- *Multimedia*
- *Application Control*

Digital Technical Journal

Digital Equipment Corporation



Volume 5 Number 2

Spring 1993

Cover Design

Dithering and color space conversion are two of the concepts discussed in "Video Rendering," which opens this issue's set of papers on multimedia technologies. On the cover, the band of blue across the bottom of the cover graphic shows the rectangular patterning created by an ordered dither process using a popular recursive tessellation array. The band of burgundy across the top shows the superior patterning of the same ordered dither process with a newly designed void-and-cluster array, which produces a higher quality image for display by eliminating the rectangular patterns and the textures of white noise. The line illustration overlaying these two arrays presents two color spaces, one within the other: RGB and YUV (luminance-chrominance space used by television systems; Y axis not shown). In the color conversion process, data transmitted in YUV space is converted to RGB space. The cover design shows three faces of the RGB space "lifted off" and infused with the colors noted at each corner of the parallelepiped.

The cover concept and illustrations are derived from the paper "Video Rendering" by Bob Ulichney. The design was implemented by Linda Falvella of Quantic Communications, Inc.

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@CRL.DEC.COM. Single copies and back issues are available for $16.00 each from Digital Press of Digital Equipment Corporation, 129 Parker Street, Maynard, MA 01754. Recent back issues of the *Journal* are also available on the Internet at gatekeeper.dec.com in the directory /pub/DEC/DECinfo/DTJ.

Digital employees may send subscription orders on the ENET to RDVAX::JOURNAL. Orders should include badge number, site location code, and address.

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. Digital Equipment Corporation assumes no responsibility for any errors that may appear in the *Journal*.

ISSN 0898-901X

Documentation Number EY-P963E-DP

Book production was done by Quantic Communications, Inc.

Contents

Multimedia

Application Control

Editor's Introduction

Jane C. Blake
Managing Editor

This issue of the *Digital Technical Journal* features papers on multimedia technologies and applications, and on uses of the Application Control Architecture (ACA), Digital's implementation of the Object Management Group's CORBA specification.

The high quality of today's television, film, and sound recordings have set expectations for computer-based multimedia; we expect high-quality images, fast response times, good quality audio, availability—including network transmission, and all at "reasonable" cost. Bob Ulichney has written about video image-rendering methods that are in fact fast, simple, and inexpensive to implement. He reviews a color rendering system and compares techniques that address the problem of insufficient colors for displaying video images. Dithering is one of these techniques, and he describes a new algorithm which provides good quality color and high-speed image rendering.

The dithering algorithm is utilized in Software Motion Pictures. SMP is a method for generating digital video on desktop systems without the need for expensive decompression hardware. Burkhard Neidecker-Lutz and Bob Ulichney discuss issues encountered in designing portable video compression software to display digital video on a range of display types. SMP has been ported to Alpha AXP, Sun, IBM, Hewlett-Packard, and Microsoft platforms.

Digitized data—video or audio—must be compressed for efficient storage and transmission. Davis Pan surveys audio compression techniques, beginning with analog-to-digital conversion and data compression. He then discusses the Motion Picture Experts audio algorithm and the interesting problem of developing a real-time software implementation of this algorithm.

Even compressed, digitized data takes up tremendous amounts of storage space. A relational database can not only store this data but provide fast retrieval. Mark Riley, Jay Feenan, John Janosik, and T.K. Rengarajan describe DEC Rdb enhancements that support multimedia objects, i.e., text, still frame images, compound documents, and binary large objects.

Managing image documents is the subject of a paper by Jan te Kiefte, Bob Hasenaar, Joop Mevius, and Theo van Hunnik. Megadoc is a hardware and software framework for building customized image management applications quickly and at low cost. They describe the UNIX file system interface to WORM drives, a storage manager, and an image application framework.

Distributing multimedia over a network presents both engineering challenges and opportunities for applications. DECspin is a real-time, desktop video-conferencing application that operates over LANs or WANs, using TCP/IP or DECnet protocols. Larry and Ricky Palmer present an overview of the DECspin graphical interface. They then address network issues of real-time conferencing on non-real-time networks and a solution to network congestion.

The transmission of full-motion video programs to multiple users requires adaptations in many parts of a client-server, LAN environment. Peter Hayden's paper focuses on the specific problem of efficient allocation of network addresses for the transmission of digital video data on a LAN. He reviews alternatives and describes a technique for the dynamic allocation of multicast addresses.

The common theme of the two final papers is ACA Services, Digital's implementation of the OMG's Common Object Request Broker Architecture. Paul Patrick has written an instructive paper on CASE environment development utilizing ACA. Assuming a multivendor, distributed environment, he discusses modeling of applications, data, and operations; application interfacing; and environment management.

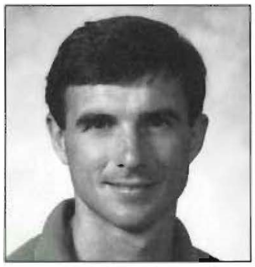
DEC @aGlance software is an implementation of ACA that supports the integration of manufacturing process information systems. David Ascher differentiates between generic integration software and @aGlance, and describes how ACA is used to integrate independently developed applications.

The editors thank John Morse, engineering manager, Corporate Research, and Mary Ann Slavin, engineering manager, ACA, for their help in preparing this issue.

Jane Blake

Biographies

David Ascher Dave Ascher joined Digital's Industrial Products Software Engineering Group in 1977 to work on the DECDataway industrial multidrop network. Since then, he has worked on distributed manufacturing systems as a developer, group leader, and technical consultant, and as an architect of the DEC @aGlance product. As a principal software engineer, Dave leads an effort to develop DEC @aGlance service offerings. He holds a B.S. in psychology from City College of New York and a Ph.D. in psychology from McMaster University, Hamilton, Ontario.



James J. Feenan, Jr. Principal engineer Jay Feenan has been implementing application code on database systems since 1978. Presently a technical leader for the design and implementation of stored procedures in DEC Rdb version 6.0, he has contributed to various Rdb and DBMS projects. Prior to joining Digital in 1984, he implemented Manufacturing Resource Planning systems and received American Production and Inventory Control Society certification. Jay holds a B.S. from Worcester Polytechnic Institute and an M.B.A. from Anna Maria College. He is a member of the U.S. National Rowing Team.

Robert Hasenaar Bob Hasenaar is an engineering manager for the Megadoc optical file system team, part of Digital's Workgroup Systems Software Engineering Group in Apeldoorn, Holland. He has seven years' software engineering experience in operating systems and image document management systems. Bob was responsible for the implementation of the first Megadoc optical disk file system in a UNIX context. He has an M.Sc. degree in theoretical physics from the University of Utrecht, Holland.

Peter C. Hayden Peter Hayden is an engineer in the Windows NT Systems Group. He joined Digital in 1986 as a member of the FDDI team and led several efforts contributing to the development of the FDDI technology and product set. He then led the Personal Computing Systems Group's multimedia projects before joining the Windows NT Systems Group in 1992. Before coming to Digital, Peter worked on PBX development at AT&T Bell Laboratories. He holds a B.S. in electrical engineering and an M.S. in computer science from Union College in Schenectady, NY, and has several patent applications pending.

John L. Janosik, Jr. A principal software engineer, John Janosik was the project leader for DEC Rdb version 5.0, the Alpha AXP port version. John has been a member of the Database Systems Group since joining Digital in 1988. Prior to this, he was a senior software engineer for Wang Laboratories Inc. and worked on PACE, Wang's relational database engine and application development environment. John received a B.S in computer science from Worcester Polytechnic Institute in 1983.

Joop W. Mevius A systems architect for the Megadoc system, Joop Mevius has over 25 years' experience in software engineering. He has made contributions in the areas of database management systems, operating systems, and image document management systems. Joop has held both engineering management positions and technical consultancy positions. He has an M.Sc. degree in mathematics from the Technical University of Delft, Holland.

Burkhard K. Neidecker-Lutz Burkhard Neidecker-Lutz is a principal engineer in the Distributed Multimedia Group of Digital's Campus-based Engineering Center in Karlsrühe. He currently works on distributed multimedia services for broadband networks. Burkhard contributed to the XMedia layered product. Prior to that work he led the design of the NESTOR distributed learning system. He joined Digital in 1988 after working for PCS computer systems. Burkhard earned an M.S. in computer science from the University of Karlsrühe in 1987.

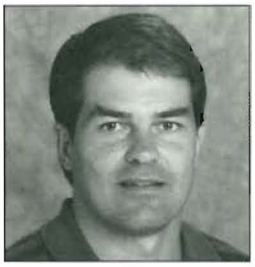
Lawrence G. Palmer Larry Palmer is a principal engineer with the Networks Engineering Architecture Group. He currently leads the DECspin project for the PC and has been with Digital since 1984. Larry is one of three software developers who initiated the PMAX software project for the DECstation 3100 product by porting the ULTRIX operating system to the MIPS architecture. He has a B.S. (highest honors, 1982) in chemistry from the University of Oklahoma and is a member of Phi Beta Kappa. He is co-inventor for five patents pending on enabling software technology for audio-video teleconferencing.

Ricky S. Palmer Ricky Palmer is a principal engineer with the Computer Systems Group. He joined Digital in 1984 and currently leads the DECspin project. Ricky is one of three software developers who initiated the PMAX software project for the DECstation 3100 product by porting the ULTRIX operating system to the MIPS architecture. He has a B.S. (high honors, 1980) in physics, a B.S. (1980) in mathematics, and an M.S. (1982) in physics, all from the University of Oklahoma. He is co-inventor for five patents pending on enabling software technology for audio-video teleconferencing.

Davis Yen Pan Davis Pan joined Digital in 1986 after receiving a Ph.D. in electrical engineering from MIT. A principal engineer in the Alpha Personal Systems Group, he is responsible for the development of audio signal processing algorithms for multimedia products. He was project leader for the Alpha/OSF base audio driver. He is a participant in the Interactive Multimedia Association Digital Audio Technical Working Group, the ANSI X3L3.1 Technical Working Group on MPEG standards activities, and the ISO/MPEG standards committee. Davis is also chair of the ISO/MPEG ad hoc committee of MPEG/audio software verification.

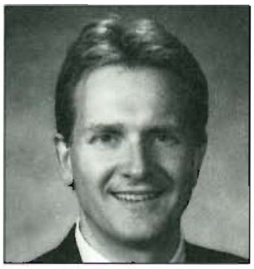
Paul B. Patrick, Sr. Paul Patrick is a principal software engineer in the ACA Services Group. He leads Digital's implementation of the Object Management Group's Common Object Request Broker Architecture. Previously, Paul helped design COHESION, an integrated CASE environment based on the DECset architecture. He also contributed to the development of IPSE, an integrated project support environment based on the CDD/Repository software, and designed and implemented the MicroVAX 2000 synchronous controller diagnostic. Prior to joining Digital, Paul held positions at GenRad Inc. and Norand Corp.

T. K. Rengarajan T. K. Rengarajan, a member of the Database Systems Group since 1987, works on the KODA software kernel of the DEC Rdb system. He has contributed in the areas of buffer management, high availability, OLTP performance on Alpha AXP systems, and multimedia databases. Presently, he is working on high-performance logging, recoverable latches, asynchronous batch writes, and asynchronous prefetch for DEC Rdb version 6.0. Ranga holds M.S. degrees in computer-aided design and computer science from the University of Kentucky and the University of Wisconsin, respectively.



Mark F. Riley Consulting software engineer Mark Riley has been a member of the Database Systems Group since 1989 and works on multimedia data type extensions in Rdb/VMS. Prior to this, he worked for five years in the Image Systems Group and developed parts of the DECimage Application Services toolkit. Mark received a B.S.E.E. from Worcester Polytechnic Institute in 1980 and an M.S. in engineering from Dartmouth College in 1982.

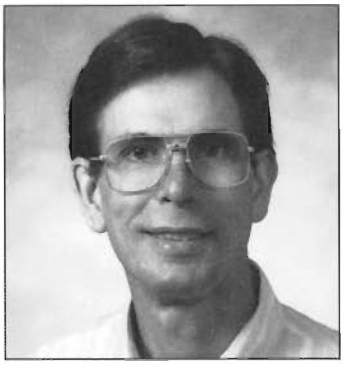
Jan B. te Kiefte Jan te Kiefte is technical director for Digital's Workgroup Systems Software Engineering Group in Apeldoorn, Holland. He has over 20 years' software engineering experience in compiler development and in the development of office automation products. Jan has held both engineering management positions and technical consultancy positions. He has an M.Sc. degree in mathematics from the Technical University of Eindhoven, Holland.



Robert Ulichney Robert Ulichney received his Ph.D. (1986) in electrical engineering and computer science from the Massachusetts Institute of Technology and his B.S. (1976) in physics and computer science from the University of Dayton, Ohio. He is a consulting engineer in Alpha Personal Systems, where he manages the Codecs and Algorithms Group. Bob has nine patents pending for his contributions to a variety of Digital products, is the author of *Digital Halftoning,* published by The MIT Press, and serves as a referee for several technical societies including IEEE.

Theo M. van Hunnik Theo van Hunnik is an engineering project manager for Digital's Workgroup Systems Software Engineering Group in Apeldoorn, Holland. He has over 20 years' software engineering experience in compiler development and the development of office automation products. Theo has participated in several international systems architecture task forces. He managed the development team for RetrievAll, the Megadoc image application framework.

Foreword



John A. Morse
Sr. Engineering Manager, Corporate Research & Architecture

In the late '80s, "multimedia" was a magic word. It seduced us with glimpses of a brave new world where audio and video technology merged with computer technology. It promised us everything from instant high-impact business presentations to virtual reality. Words like "paradigm shift" and "multibillion-dollar industry" were enough to snare both the technophiles and the eager entrepreneurs into believing that the world had suddenly changed, and we were all going to get rich in the process.

Somewhere on the way to the bank, reality set in, and it wasn't virtual. The reality is that multimedia is a lot harder than it looks. Successful multimedia requires a marriage between analog TV technology and digital computer technology; it requires reconciliation between a technical/professional marketplace and a consumer marketplace. As in any marriage, a lot of hard work is required to make it succeed, and much of that work is yet to be done.

For certain segments of the computer industry, multimedia was relatively easy to implement and so caught on quickly. The first successes have been at the extremes of the cost spectrum—very low-end desktop multimedia on the one hand, and very high-end virtual reality systems on the other. This has left Digital, with its traditional focus on the middle, temporarily out of the game.

For desktop multimedia, all that is required is the ability to capture and display video and audio. Since machines like the Commodore Amiga were already based more on TV technology than on computer technology (for cost reasons), they could be quickly and cheaply adapted to handle audio and motion video. Thus desktop multimedia was born. The CD-ROM, adapted from audio CD technology, was the perfect storage medium for distribution of multimedia content; and so for this market segment, CD-ROM and multimedia became almost synonymous. There has emerged a whole industry based around the production of multimedia titles on CD-ROM.

At the high end, for purposes such as full-realism aircraft simulation or virtual reality applications, the solution was to use the highest performance hardware available, at whatever expense. Typically, high-end, three-dimensional graphics systems were coupled either to supercomputers or to massively parallel processor arrays. The result was, and still is, impressive. But the cost is still so high that such virtual reality systems are not yet commercially viable except in specialized low-volume markets.

The vast area in the middle, into which all of Digital's business falls, has developed very slowly. The problem is that our business is based on a model of enterprise-wide computation. The computer systems we design and sell not only include processors and displays but incorporate networks and servers as well. To introduce multimedia into such a model, one touches every aspect of the system, from the desktop, through the network, and back to the servers. At every turn, we have found that the technology that has evolved over 30 to 40 years for handling numbers, text, and (more recently) two-dimensional and three-dimensional graphics is not quite right for video and audio. Every component of the system, both hardware and software, needs to change in some way. We need to evolve to a model of networked client-server multimedia computing. Change of this magnitude is a slow process.

Two challenges are so pervasive that almost every paper in this issue addresses them, each from a different perspective. First of all, multimedia involves the handling of large quantities of data. Second, for many applications, that data must be handled under very tight time constraints. The resulting stress and strain on all components of the system translates into a set of technical challenges that has occupied us for the last four years and promises to keep us busy through at least the rest of this decade.

Depending on the picture quality chosen, it may require from one million to one hundred million bytes of st●rage to save each *second* of live video in digital form. Since many applications of multimedia, such as archiving television footage for research or historic preservation purposes, will need to save many *hours* of video, it is easy to see that multimedia quickly builds demand for many gigabytes (1,000,000,000 bytes) of magnetic or optical disk storage. But storage is only part of the problem. Once such enormous amounts of data are stored, the challenge becomes how to retrieve a particular item of interest. Standard database techniques are oriented toward retrieval of text and numbers. Retrieval of audio and video information will require new file and database techniques that are only beginning to be understood.

An obvious application of multimedia technology, once the networks are in place, is teleconferencing. We can envision a day when we can connect to anyone any place in the world via the network and carry on a conversation with them, while each of us sees the other in full-motion video, using the audio and video capabilities of our desktop workstations and PCs. But realizing this vision has proved surprisingly hard. People expect the images they see to be synchronized with the sounds they hear, and they expect delays to be no worse than those experienced on a long-distance telephone call. Unfortunately, data networks have been designed to maximize throughput and reliability. They do this at the expense of some delay in transmission—delay that is annoying at best, and unacceptable at worst, for teleconferencing applications.

Successful infusion of multimedia technology into enterprise-wide computation is proving to require change on a scale that almost no one anticipated. We at Digital are in the midst of this process of change, and this issue of the *Digital Technical Journal* is a snapshot, taken at one point in time, of that process. Together, the papers describe some of the toughest technical challenges that we face and in many cases give glimpses into possible solutions.

Robert Ulichney

Video Rendering

Video rendering, the process of generating device-dependent pixel data from device-independent sampled image data, is key to image quality. System components include scaling, color adjustment, quantization, and color space conversion. This paper emphasizes methods that yield high image quality, are fast, and yet are simple and inexpensive to implement. Particular attention is placed on the derivation and analysis of new multilevel dithering schemes. While permitting smaller frame buffers, dithering also provides faster transport of the processed image to the display—a key benefit for the massive pixel rates associated with full-motion video.

Perhaps the most influential characteristic governing the perceived value of a system that displays images is the way the pictures look. Image appearance is largely dependent upon the quality of rendering, that is, the process of taking device-independent data and generating device-dependent data tailored for a particular target display.

The topic of this paper is the processing of sampled image data and not synthetic graphics. For graphics rendering, primitives such as specifications of triangles are converted to displayable picture elements or pixels. The atomic elements handled by a video rendering system are device-independent pixels. Whereas a prerendered graphics image can be compactly represented as a collection of triangle vertices, prerendered video achieves compaction by means of compression techniques.

Sampling broadcast video requires a data rate of more than 9 million color pixels per second; the need of some relief for storage and networks is clear. Video compression reduces redundancy in the source image and thereby reduces the amount of data to be transmitted. Dramatic reductions in data rate can be achieved with little degradation in image quality. The Joint Photographic Experts Group (JPEG) standard for still frame and the Motion Picture Experts Group (MPEG) and Px64 standards for motion video are current committee compression techniques.[1] Several other nonstandard schemes exist, including a simple compression method conducive to software-only implementation.[2]

Video rendering receives decompressed image data as input. Since every decompressed pixel must be processed, speed is essential. This paper focuses on rendering methods that are fast, simple, and inexpensive to implement. Performance at video rates can be achieved with minimal hardware or even software-only solutions.

The Rendering Architecture section reviews the components of a rendering system and examines design trade-offs. The paper then presents details of new and efficient dithering implementations. Finally, video color mapping is discussed.

Rendering Architecture

Figure 1 illustrates the major phases of a video rendering system: (1) filter and scale, (2) color adjust, (3) quantize, and (4) color space convert.

In the first stage, the original image data must be resampled to match the target window size. A separate scaling system should be used for the horizontal and vertical directions to handle the case where the pixel aspect ratio must be changed. For example, such asymmetric scaling is needed when the target display expects square pixels and the original pixels are not square.

The best filters to use in combination with scaling have been determined from a perceptual point of view.[3] When limiting the bandwidth to reduce the data rate, a Gaussian filter with a standard deviation $\sigma = 0.30 \times$ output period is recommended. For interpolation, the filter preferred (because the filtered results looked most like the original) was a cascade of two: first, sharpen with a Laplacian filter, and second, follow by convolution with a Gaussian filter with $\sigma = 0.375 \times$ input period.

A typical sharpening scheme can be expressed by the following equation:

$$I_{\text{sharp}}[x,y] = I[x,y] - \beta\Psi[x,y] * I[x,y], \qquad (1)$$

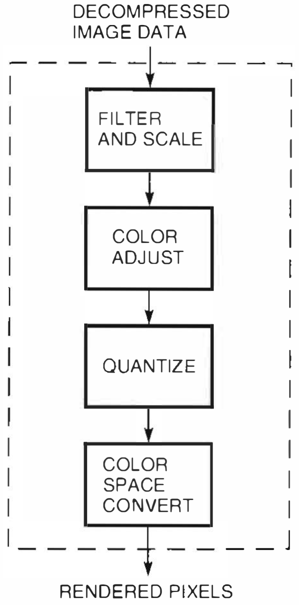


Figure 1 Image Rendering System

where $I[x,y]$ is the input image, $\Psi[x,y]$ is a digital Laplacian filter, and "$*$" is the convolution operator.[4] The nonnegative parameter β controls the degree of sharpness, with $\beta = 0$ indicating no change in sharpness. When enlarging, sharpening should occur before scaling, and when reducing, sharpening should take place after scaling. The filtering discussed here is assumed to be two-dimensional, which requires image line buffering. For economy, horizontal-only filtering is sometimes used.

The simplest means of scaling is known as nearest-neighbor scaling, and its simplest implementation is based on the Bresenham scan conversion algorithm for drawing straight lines.[5] This algorithm can be applied to image scaling and performed with only three registers and one adder.[6] Further optimizations make this algorithm especially suited for real-time use.[7]

The second stage of rendering is color adjust, most easily achieved with a look-up table (LUT). Each color component uses a separate adjust LUT. In the case of a luminance-chrominance color, an adjust LUT for the luminance component controls contrast and brightness, and LUTs for the chrominance components control saturation.

For so-called true-color frame buffers with 24-bit depths, visual artifacts that can result from insufficient amplitude resolution do not occur. With smaller frame buffers, restricting the amplitude of the color components red, green, and blue (RGB) with a simple uniform quantizer causes false contours to appear in slowly varying regions. This issue leads to the third stage in the rendering system, quantization.

The three basic classes of techniques for circumventing the problem of insufficient colors or color memory are (1) histogram-based methods, (2) chrominance-subsampled frame buffers, and (3) dithering. All histogram-based methods, sometimes called palette selection, require two passes of the entire image data: the first to acquire the histogram statistics to fabricate a three-dimensional quantizer to N colors and the second to perform the pixel assignments. Perhaps the fastest method is the popularity algorithm, where a simple sort finds the N colors with the highest frequency, and all other colors are mapped to those.[8]

A more compute-intensive method, but one that in general performs much better, is the often-used, median-cut algorithm.[8] In this method, the color space is repeatedly subdivided into smaller rectangular solids at the median planes, with the goal that each of the selected colors represent an equal number of colors in the image. The average of the colors in each of the final regions is the color used in the quantizer. A later, less compute-intensive variation is the mean-split algorithm. Also, several clustering techniques have been reported that result in less quantization error than the above-mentioned methods. One method, for example, minimizes the sum of the squares of the errors.[9] In all cases, however, color problems can occur in other application windows because each frame requires a different color map; the colors in the other windows become scrambled in a different way for each color map.

One advantage of representing image data in a luminance-chrominance space is that chrominance requires less spatial resolution than luminance to achieve excellent image quality. Visual perception of differences in chrominance is much less than that for luminance. The television standards have been exploiting this fact for decades. The quantization approach of using chrominance-subsampled frame buffers is built on this fact, deferring conversion to the RGB components until just after the data is read for display.[10,11,12]

Typical implementations of chrominance-subsampled frame buffers average each of the two chrominance values in a given luminance-chrominance color representation over a region that is either 2 by 2 or 4 by 4 pixels. Assuming 8 bits of amplitude resolution per color component, the 2-by-2-pixel case results in an average of ((2 × 2 × 8 luminance bits) + (8 + 8 chrominance bits))/(2 × 2 pixels) or 12 bits per pixel; similarly, the 4-by-4-pixel case results in 9 bits per pixel. This approach requires expensive hardware to up-sample the chrominance components and convert the color space at video rates. These nonstandard frame buffers can also cause severe incompatibility problems with most applications that expect RGB frame buffers. While chrominance subsampled frame buffers can accommodate most sampled natural images, thin-line graphics can be annihilated.

The third alternative for quantization is to use a dithering method. Several methods exist that are designed primarily for binary output, but all are extendable to multilevel color.[4,13,14,15] A "level" is a shade of gray, from black to white, or a shade of a color component, from black to the brightest value. The basic principle of dithering is to use the available subset of colors to produce, by judicious arrangement, the illusion of any color in between.

Although neighborhood operations, most notably error diffusion, produce good-quality dithering, they are computationally complex and require additional storage. For video processing, where speed is essential, we turned our focus to those dithering methods that are point operations, that is, methods that operate on the current pixel only without considering its neighbors. Each color component of every pixel in the image has an associated "noise" or dither amplitude that is added to it before that component is passed to a uniform quantizer.

Historically, the first dithering method used for video processing was white noise dithering, where a pseudorandom number was added to each luminance value before quantization. This method was practiced soon after the dawn of television.[16] However, the low-frequency energy in white noise causes undesirable textures and graininess.

A preferred method is the point process of ordered dithering, where a deterministic noise array tiles the plane in a periodic manner. Dither arrays can be designed to minimize low-frequency texture. The most popular are the so-called recursive tessellation arrays.[17,18] These arrays yield results superior to those of white noise dithering but suffer from structured rectangular patterns.

A new ordered dither array design, called the "void-and-cluster" method, eliminates both the low-frequency textures of white noise and the rectangular patterns of recursive tessellation arrays.[19] The name describes the dither array design process in which voids and clusters are located and mitigated.

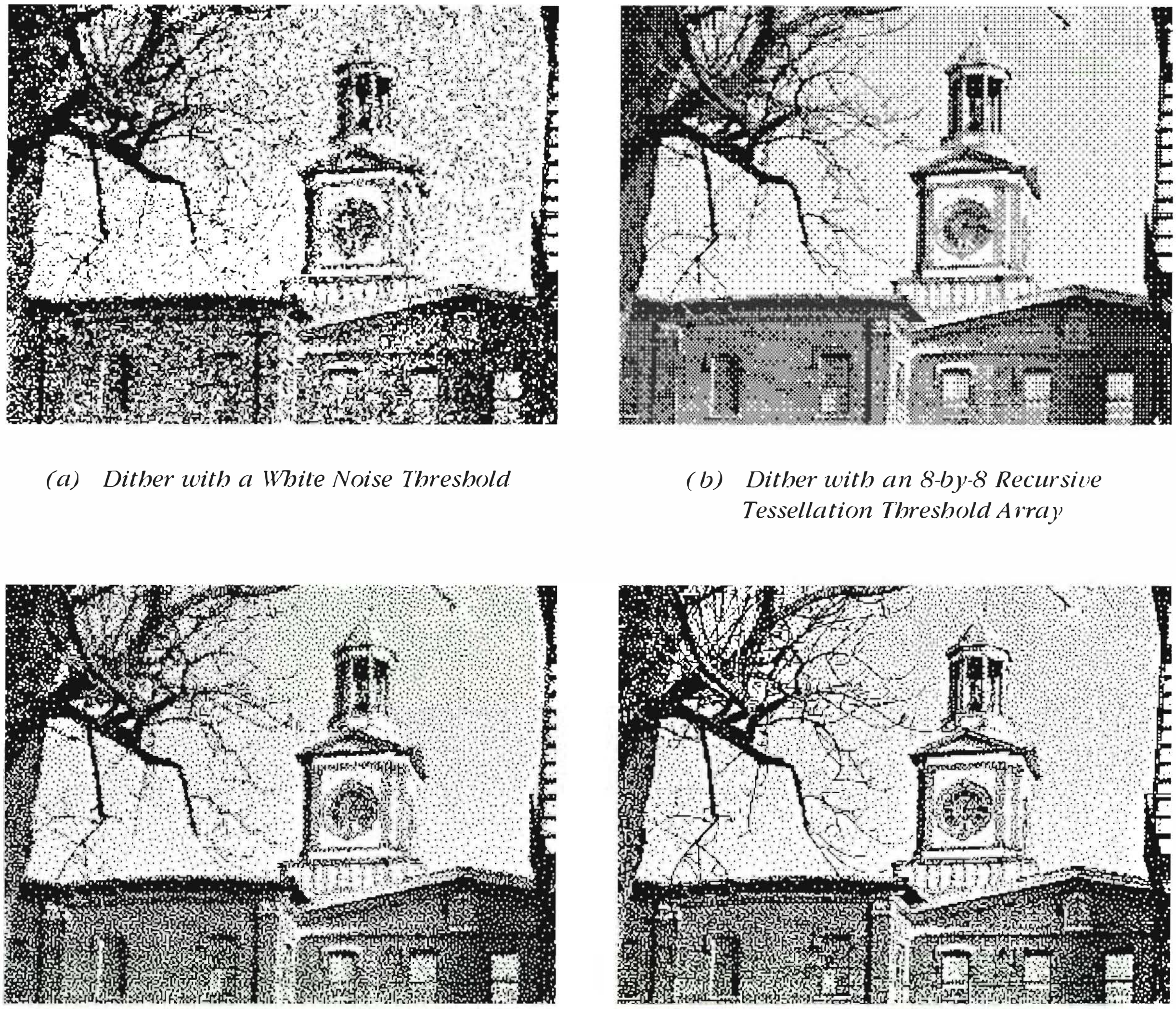
For the high-speed case of motion video, an ordered dithering scheme has important advantages over chrominance-subsampled frame buffers and histogram-based approaches. Quantization by dithering allows the use of conventional frame buffers, does not require the time-consuming process of making two passes over each frame (or every *N*th frame), does not cause other applications to change color maps with every *N*th frame, and allows any number of colors to be selected at render time. Also, experiments have shown that the image quality achieved by dithering is very competitive with the other methods, when compared over a range of sample images. Even when 24-bit frame buffers are available, the increased speed of loading three or four 8-bit color pointers or index values in the time required to load a single 24-bit pixel makes dithering a viable alternative in the design of desktop video systems.

By way of comparison, Figure 2 illustrates some of the methods described in this section. A 240-by-360-pixel, 8-bit monochrome image was rendered to only two levels and displayed at 100 dots per inch (dpi). Figure 2a depicts an image that was dithered with white noise; in Figure 2b, the same image was dithered using an 8-by-8 recursive tessellation dither array; and Figure 2c shows the image dithered with the new 32-by-32 void-and-cluster array. To illustrate the effect of sharpening, Figure 2d shows the image in Figure 2c presharpened using a digital Laplacian filter as in equation (1), with a sharpening factor of $\beta = 2.0$. The goal of this coarse example is to amplify the different effects. The same methods apply to multilevel and color output, where the resulting quality is much higher.

Fast Multilevel Dithering

This section presents the details of simple, yet powerful new designs to perform multilevel ordered dithering. The simplicity of these methods allows for implementation with minimum hardware or software only, yet guarantees output that preserves the mean of the input. The designs are flexible in that they allow dithering from any number of input levels N_i, to any number of output levels N_o, provided $N_i \geq N_o$. Note that N_i and N_o are not restricted to be powers of two.

Each color component of a color image is treated as an independent image. The input image L_i can have values



(a) Dither with a White Noise Threshold

(b) Dither with an 8-by-8 Recursive Tessellation Threshold Array

(c) Dither with a 32-by-32 Void-and-cluster Threshold Array

(d) Same as (c) with Laplacian Sharpening, β = 2.0

Figure 2 Examples of Rendering to Two Output Levels

$$L_i \in \{0,1,2,\ldots,(N_i - 1)\},$$

and the output image L_o can have values

$$L_o \in \{0,1,2,\ldots,(N_o - 1)\}.$$

A deterministic dither array of size $M \times N$ is used that is periodic and tiles the entire input image. To simplify addressing of this array, M and N should each be powers of two. A dither template defines the order in which dither values are arranged. The elements of the dither template T have values

$$T \in \{0,1,2,\ldots,(N_t - 1)\},$$

where N_t is the number of template levels, which represent the levels against which image input values are compared to determine their mapping to the output values. The dither template is thus central to determining the nature of the resulting dither patterns.

Figure 3 shows a dithering system that comprises two memories and an adder. The system takes an input level L_i at image location $[x,y]$ and produces output level L_o at the corresponding location in the dithered output image. The dither array is addressed by x' and y', which represent the low-order bits of

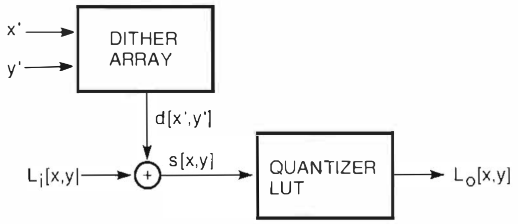


Figure 3 Dithering System with Two LUTs

the image address. The selected dither value $d[x',y']$ is added to the input level to produce the sum s. This sum is then quantized by addressing a quantizer LUT to produce the output level L_o.

The trick to achieving mean-preserving dithering is to properly generate the LUT values. The dither array is a normalized version of the dither template specified as follows:

$$d[x',y'] = \text{int}\{\Delta_d(T[x',y'] + \tfrac{1}{2})\}, \qquad (2)$$

where Δ_d, the step size between normalized dither values, is defined as

$$\Delta_d = \frac{\Delta_Q}{N_t} \qquad (3)$$

and Δ_Q is the quantizer step size

$$\Delta_Q = \frac{(N_i - 1)}{(N_o - 1)}. \qquad (4)$$

Note that Δ_Q also defines the range of dither values. The quantizer LUT is a uniform quantizer with N_o equal steps of size Δ_Q.

The precise expressions in equations (2), (3), and (4) were arrived at through extensive analysis of the average output resulting from processing input images of a constant value, over a wide range of N_i, N_o, and N_t.

One-memory Dithering System

Using the above expressions, it is possible to simplify the system by exchanging one degree of freedom for another. A bit shifter can replace the quantizer LUT at the expense of forcing the number of input levels N_i, to be set by the system. For hardware implementations, this design affords a considerable cost reduction.

The system and method of Figure 3 assume that N_i is given as a fixed parameter, as is usually the case with most imaging systems and file formats. However, for image sources such as hardware that generates synthetic graphics, arbitrarily setting N_i often has no effect on the amount of computation involved. If an adjust LUT is used to modify the image data, including a gain makes a "modified adjust LUT." Figure 4 depicts such a system, where L_r is the raw input level. The unadjusted or raw input image can have the values

$$L_r \in \{0,1,2,\ldots,(N_r - 1)\},$$

where N_r is the number of raw input levels, typically 256. Therefore, the modified adjust LUT must impart a gain of

$$\frac{N_i - 1}{N_r - 1}.$$

To solve for N_i, recall that in the method of Figure 3 the quantizer was defined to have equal steps of size Δ_Q as defined in equation (4). The quantizer LUT can be replaced by a simple R-bit shifter, if the variable Δ_Q can be forced to be an exact binary number,

$$\Delta_Q = 2^R. \qquad (5)$$

N_i can then be set by the expression

$$N_i = (N_o - 1)2^R + 1. \qquad (6)$$

The integer R is the number of bits the R-bit shifter shifts to the right to achieve quantization. Specifying R in terms of N_o, equation (6) becomes

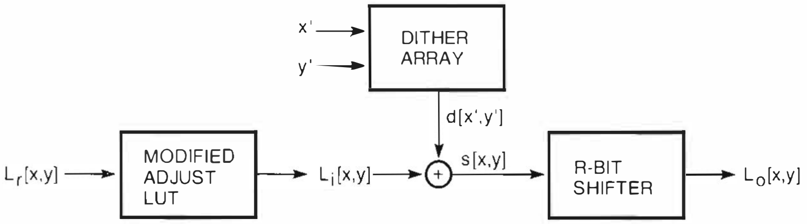


Figure 4 One-memory Dithering System with an Adjust LUT and Bit Shifter

$$R = \log_2 \frac{(N_i - 1)}{(N_o - 1)}. \tag{7}$$

To completely specify this problem requires specifying the range for N_i. It is reasonable to do this by specifying the number of bits b by which the image input values are to be represented. Specifying b limits N_i to the range

$$2^{b-1} < N_i \leq 2^b. \tag{8}$$

Parameter b will be a key value in specifying the resulting system.

Given the two expressions, (7) and (8), and the two unknowns, R and N_i, a unique solution exists because the range of N_i is less than a factor of two, and R and N_i are integers. To solve for R, substitute equation (6) for N_i in equation (8). The resulting equation is

$$2^{b-1} < (N_o - 1)2^R + 1 \leq 2^b \tag{9}$$

or

$$\log_2\left(\frac{2^{b-1} - 1}{N_o - 1}\right) < R \leq \log_2\left(\frac{2^b - 1}{N_o - 1}\right). \tag{10}$$

Since $2 \leq N_o \leq N_i$, the range of the expression in equation (10) must be less than one. Hence, given that R is an integer,

$$R = \text{int}\left\{\log_2\left(\frac{2^b - 1}{N_o - 1}\right)\right\}. \tag{11}$$

N_i in equation (6) is now specified.

As an example, consider the case where N_o equals 87 (levels), b equals 9 (bits), N_t equals 1,024 (levels, for a 32-by-32 template), and N_r equals 256 (levels). Thus, R equals 2, and the R-bit shifter drops the least-significant 2 bits. N_i equals 345 (levels); the dither array is normalized by equation (2) with $\Delta_d = 1/256$; and the gain factor to be included in the modified adjust LUT is 344/255. This data is loaded into the system represented by Figure 4 and uniformly maps input pixels across the 87 true output levels, giving the illusion of 256 levels.

The output image that results from either of the dithering systems illustrated in Figure 3 or Figure 4 appears to contain more effective levels than are actually displayed. An effective level is either a perceived average level that is dithered between two true output levels or shades or an actual true output level. A small number of template levels N_t dictates the resulting number of effective levels. When N_t is large, the number of effective levels is equal to the number of input levels N_i, because it is not possible to display more effective outputs than inputs. More precisely,

$$\text{Effective Levels} = \begin{cases} (N_o - 1)\,N_t + 1 & \frac{\Delta_Q}{N_t} > 1 \\ N_i & \frac{\Delta_Q}{N_t} \leq 1. \end{cases} \tag{12}$$

Note that Δ_Q/N_t in equation (12) is equal to Δ_d. When $\Delta_d < 1$, the normalization of the dither array, i.e., equation (2), results in integer truncated values that are not all unique. At this point, the number of effective levels saturates to N_i.

Data Width Analysis

The design of an efficient dithering system, particularly in hardware, depends on knowing the number of bits required for all data paths in the system. This section presents an analysis of the one-memory dithering system shown in Figure 4.

The system input b, i.e., the bit depth of the image input values, limits the data path for $L_i[x,y]$. The analysis shows the derivation of the precise bit depths for the other data paths. In summary, the derivation proves that the dither values in the dither matrix memory require R bits, where $R_{max} = (b - 1)$ and $s = L_i + d$ (and thus the R-bit shifter) require only b bits.

Bits Needed for Dither Matrix The amount of memory needed to store the dither matrix is an important concern; d_{max} denotes the maximum value. To determine d_{max}, substitute the maximum value of $T[x',y']$, which is $(N_t - 1)$, into equation (2). The resulting equation is

$$\begin{aligned} d_{max} &= \text{int}\left\{\frac{2^R}{N_t}\left((N_t - 1) + \frac{1}{2}\right)\right\} \\ &= \text{int}\left\{2^R\left(\frac{N_t - \frac{1}{2}}{N_t}\right)\right\}. \end{aligned} \tag{13}$$

d_{max}, which depends on N_t, thus has a value in the range

$$2^{R-1} \leq d_{max} \leq 2^R - 1. \tag{14}$$

For the case of a dither matrix with one value, namely $N_t = 1$, d_{max} equals the lower end of this range. d_{max} equals the high end of the range for large dither matrices, where $2^{R-1} \leq N_t$. An important observation is that for all values in the range of expression (14), the number of bits needed is exactly R.

From equation (11), the value of R increases as N_o decreases. The smallest possible value of N_o is 2, which is for bitonal output. In this case, the maximum value of R is

$$R_{max} = \text{int}\{\log_2(2^b - 1)\} = (b - 1). \qquad (15)$$

So, the number of bits needed for the dither values is R, which can be as large as $(b - 1)$.

Bit Width of Adder Recall that $s[x,y] = L_i[x,y] + d[x,y]$. The number of bits needed for this sum determines the size of the adder and the size of the R-bit shifter. L_i can be at most $(N_i - 1)$ and, as determined in the last section, d_{max} can be at most $(2^R - 1)$. So,

$$s_{max} = (N_i - 1) + (2^R - 1). \qquad (16)$$

From equation (6),

$$(N_i - 1) = (N_o - 1)2^R, \qquad (17)$$

which gives

$$s_{max} = 2^R(N_o - 1) + (2^R - 1) = 2^R N_o - 1. \quad (18)$$

We can express R in terms of N_o by using equation (11):

$$R = \text{int}\{\log_2(2^b - 1) - \log_2(N_o - 1)\}. \qquad (19)$$

Each of the two terms in the equation (19) can be expressed in terms of an integer part and a fractional part:

$$\log_2(2^b - 1) = (b - 1) + \epsilon_1, \qquad (20)$$

where

$$0 < \epsilon_1 < 1,$$

and

$$\log_2(N_o - 1) = K + \epsilon_2, \qquad (21)$$

where K is an integer, and

$$0 \le \epsilon_2 < 1.$$

Now equation (19) can be rewritten as

$$R = (b - 1) - K + \text{int}\{\epsilon_1 - \epsilon_2\}. \qquad (22)$$

ϵ_2 is largest when N_o (an integer) is a large power of 2. Because N_o cannot be greater than N_i,

$$2^b \ge N_o.$$

This fact, combined with equations (20) and (21), yields the further condition

$$\epsilon_1 \ge \epsilon_2.$$

Therefore, $\text{int}\{\epsilon_1 - \epsilon_2\}$ in equation (22) must be equal to zero, and the value of R becomes

$$R = b - 1 - K. \qquad (23)$$

We can express N_o in equation (18) in terms of the same integer K of equation (21) by noting that

$$\log_2 N_o = K + \epsilon_3, \qquad (24)$$

where

$$0 < \epsilon_3 \le 1. \qquad (25)$$

Observe that ϵ_3 is equal to 1, where N_o is an exact power of 2. Substituting

$$N_o = 2^{K+\epsilon_3}$$

and equation (23) into equation (18) gives

$$s_{max} = 2^{b-1-K}\,2^{K+\epsilon_3} - 1 = 2^{b-1+\epsilon_3} - 1. \quad (26)$$

Because of the range of ϵ_3 in equation (25), the range of s_{max} must be

$$2^{b-1} - 1 < s_{max} \le 2^b - 1, \qquad (27)$$

which requires exactly b bits.

As a check, the size of the shift register should equal the number of bits required for N_o plus R. The number of bits needed for N_o is

$$\text{int}\{1 + \log_2(N_o - 1)\}. \qquad (28)$$

Using the expression in equation (21), this value becomes

$$\text{int}\{1 + K + \epsilon_2\} = K + 1. \qquad (29)$$

So, the size of the shift register must be

$$(K + 1) + (b - 1 - K) = b \text{ bits},$$

which matches the maximum size of the sum s.

Color Space Conversion

Referring once again to Figure 1, consider the final subsystem of a video rendering system—color space convert. Assuming a frame buffer that is expecting RGB data, color space conversion is not necessary if the source data is already represented in RGB, as in the case of graphics generation systems. However, motion video is essentially always transmitted and stored in a luminance-chrominance space. Such a representation allows subsampling of the chrominance, as mentioned earlier, which reduces bandwidth requirements; all video standards exploit this method of bandwidth reduction. It is also more intuitive to color adjust in a luminance-chrominance space.

Prior to proceeding to the quantize subsystem shown in Figure 1, all color components must be at the same final spatial resolution for a dithering method to work correctly. Chrominance components, then, need to be up-sampled to the same rate as luminance components.

Although the chromaticities of the RGB primaries of the major television standards vary slightly, all television systems transmit and store the color data in YUV space. Y represents the achromatic component that is loosely called the luminance component. (The term luminance has a specific photometric definition that is not what is represented in a video Y component.) U and V are color difference components, where U is proportional to (Blue − Y) and V is proportional to (Red − Y).

Figure 5 is an orthographic projection of YUV space. Inside the YUV rectangular solid is the

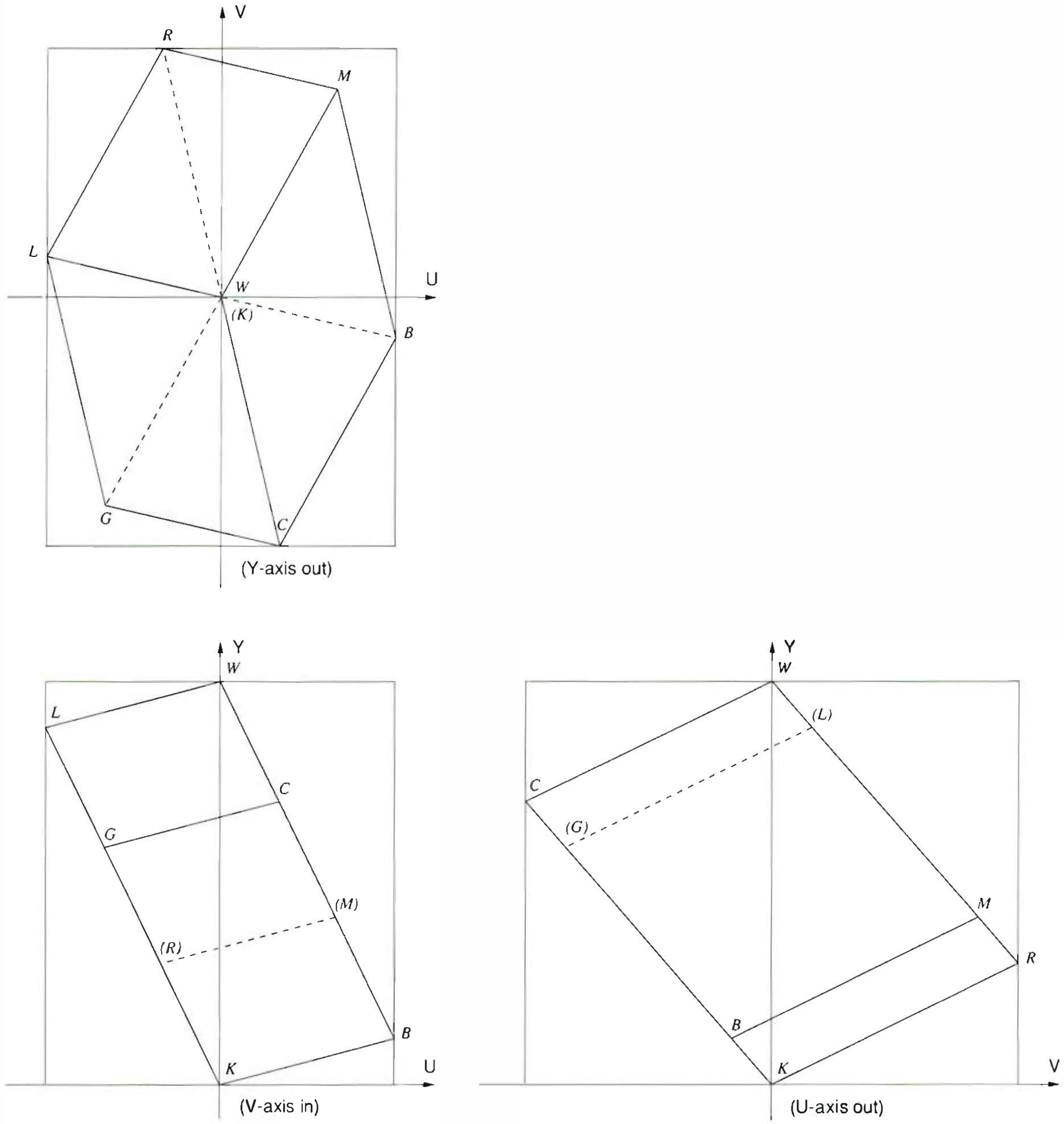


Figure 5 Feasible RGB Values in the YUV Color Space

parallelepiped of "feasible" RGB space. Feasible RGB points are those that are nonnegative and are not greater than the maximum supported value. For reference, the corners of the RGB parallelepiped are labeled black (K), white (W), red (R), green (G), blue (B), cyan (C), magenta (M), and yellow (L). RGB and YUV values are related linearly and can be interconverted by means of a 3-by-3 matrix multiply.

In the United States video broadcast system, the chrominance plane (i.e., the U-V plane in Figure 5) is rotated 33 degrees by introducing a phase in the quadrature modulation of the chrominance signal. The resulting rotated chrominance signals are renamed I and Q (for inphase and quadrature), but the unmodulated color space is still YUV.

Figure 6 shows the back end of a rendering system that uses dithering as a quantization step prior to color space conversion. A serendipitous consequence of dithering is that color space conversion can be achieved by means of table look-up. The collective address formed by the dithered Y, U, and V values is small enough to require a reasonably sized color mapping LUT. There are two advantages to this approach. First, a costly dematrixing operation is not required, and second, infeasible RGB values can be intelligently mapped back to feasible space off-line during the generation of the color mapping LUT.

This second advantage is an important one, because 77 percent of the valid YUV coordinates are in invalid RGB space, i.e., the space around the RGB parallelepiped in Figure 5. Color adjustments such as increasing the brightness or saturation can push otherwise valid RGB values into infeasible space. In alternative systems that perform color conversion by dematrixing, out-of-bounds RGB values are simply truncated. This operation effectively maps colors back to feasible RGB space along lines perpendicular to a parallelepiped surface illustrated in Figure 5, which can change the color in an undesirable way. The use of a color mapping LUT avoids these problems.

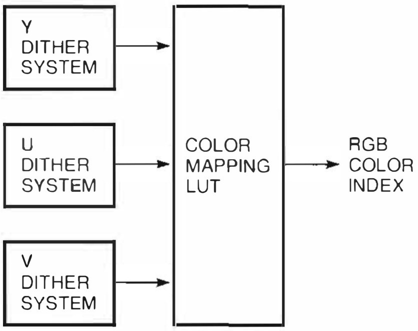


Figure 6 System for Dithering Three-color Components and Color Mapping the Collective Result

Summary

Video is becoming an increasingly important data type for desktop systems. This is especially true as distinctions between computing, consumer electronics, and communications continue to blur. While many factors contribute to the impression one has of the value of a product that displays information, the way the images look can make the biggest difference. This paper focuses on rendering system designs that are fast, low cost, produce good-quality video, and are conducive to hardware or software implementation.

References

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

2. B. Neidecker-Lutz and R. Ulichney, "Software Motion Pictures," *Digital Technical Journal,* vol. 5, no. 2 (Spring 1993, this issue): 19-27.

3. W. Schreiber and D. Troxel, "Transformation between Continuous and Discrete Representation of Images: A Perceptual Approach," *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI),* vol. PAMI-7, no. 2 (1985): 178-186.

4. R. Ulichney, *Digital Halftoning* (Cambridge, MA: The MIT Press, 1987).

5. J. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems Journal,* vol. 4, no. 1 (1965): 25-30.

6. F. Glazer, "Fast Bitonal to Grayscale Image Scaling," DEC-TR-505 (Maynard, MA: Digital Equipment Corporation, June 1987).

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

8. P. Heckbert, "Color Image Quantization for Frame Buffer Display," *Computer Graphics AMC SIGGRAPH '82 Conference Proceedings,* vol. 16, no. 3 (1982): 297-307.

9. S. Wan, K. Wong, and P. Prusinkiewicz, "An Algorithm for Multidimensional Data Clustering," *ACM Transactions on Mathematical Software,* vol. 14, no. 2 (1988): 153-162.

10. C. Sigel, R. Abruzzi, and J. Munson, "Chromatic Subsampling for Display of Color Images," *Optical Society of America Topical Meeting on Applied Vision,* 1989 Technical Digest Series, vol. 16 (1989): 158-161.

11. A. Luther, *Digital Video in the PC Environment* (New York, NY: Intertext Publications, McGraw-Hill, 1989): 193-194.

12. L. Glass, "Digital Video Interactive," *Byte* (May 1989): 284.

13. P. Roetling, "Binary Approximation of Continuous-tone Images," *Photographic Science and Engineering,* vol. 21 (1977): 60-65.

14. J. Stoffel and J. Moreland, "A Survey of Electronic Techniques for Pictorial Reproduction," *IEEE Transactions on Communications,* vol. 29 (1981): 1898-1925.

15. J. Jarvis, C. Judice, and W. Ninke, "A Survey of Techniques for the Display of Continuous-tone Pictures on Bilevel Displays," *Computer Graphics and Image Processing,* vol. 5 (1976): 13-40.

16. W. Goodall, "Television by Pulse Code Modulation," *Bell Systems Technical Journal,* vol. 30 (1951): 33-49.

17. B. Bayer, "An Optimum Method for Two Level Rendition of Continuous-tone Pictures, *Proceedings of the IEEE International Conference on Communications, Conference Record* (1973): (26-11)-(26-15).

18. R. Ulichney, "Frequency Analysis of Ordered Dither," *Proceedings of the Society of Photo-optical Instrumentation Engineers (SPIE),* vol. 1079 (1989): 361-373.

19. R. Ulichney, "The Void-and-cluster Method for Dither Array Generation," *The Society for Imaging Science and Technology/Symposium on Electronic Imaging Science and Technology (IS&T/SPIE)* (February 1993).

Burkhard K. Neidecker-Lutz
Robert Ulichney

Software Motion Pictures

Software motion pictures is a method of generating digital video on general-purpose desktop computers without using special decompression hardware. The compression algorithm is designed for rapid decompression in software and generates deterministic data rates for use from CD-ROM and network connections. The decompression part offers device independence and integrates well with existing window systems and application programming interfaces. Software motion pictures features a portable, low-cost solution to digital video playback.

The necessary initial investment is one of the major obstacles in making video a generic data type, like graphics and text, in general-purpose computer systems. The ability to display video usually requires some combination of specialized frame buffer, decompression hardware, and a high-speed network.

A software-only method of generating a video display provides an attractive way of solving the problems of cost and general access but poses challenging questions in terms of efficiency. Although several digital video standards either exist or have been proposed, their computational complexity exceeds the power of most current desktop systems.[1] In addition, a compression algorithm alone does not address the integration with existing window system hardware and software.

Software motion pictures (SMP) is both a video compression algorithm and a complete software implementation of that algorithm. SMP was specifically designed to address all the issues concerning integration with desktop systems. A typical application of SMP on a low-end workstation is to play back color digital video at a resolution of 320 by 240 pixels with a coded data rate of 1.1 megabits per second. On a DECstation 5000 Model 240 HX workstation, this task uses less than 25 percent of the overall machine resources.

Together with suitable audio support (audio support is beyond the scope of this paper), software motion pictures provides portable, low-cost digital video playback.

The SMP Product

Digital supplies SMP in several forms. The most complete version of SMP comes with the XMedia Toolkit. This toolkit is primarily designed for developers of multimedia applications who include the SMP functionality inside their own applications. Figure 1 shows the user controls as displayed on a workstation screen. SMP players are also available on Digital's freeware compact disc (CD) for use with Alpha AXP workstations running the DEC OSF/1 AXP operating system. In addition, SMP playback is included with several Digital products such as the video help utility on the SPIN (sound picture information networks) application, as well as other vendors' products, such as the MediaImpact multimedia authoring system.[2]

In the XMedia Toolkit, access to the SMP functions is possible through X applications, command line utilities, and C language libraries. The applications and utilities support simple editing operations, frame capture, compression, and other functions. Most of these features are intended for use by producers of simple file formats called SMP clips.

The decompression functionality is offered as an X toolkit widget that readily integrates into the Open Software Foundation's (OSF) Motif-based applications. Multiple SMP codecs (compressors/decompressors) on a given screen all share the same color resources with one another and with the Display PostScript X-server extension, which is offered by all major workstation vendors. It also plays well with the standard color allocations used in the Macintosh QuickDraw rendering system and Microsoft Windows standard color allocations.

To facilitate flexible but simple access to entire films of SMP frames, SMP defines SMP clips. Rather than publishing that file format directly, all applications and widgets are accessed through an encapsulating library. This method allows future releases to have application-transparent changes to the underlying file structure and completely different ways to store and obtain SMP frames.

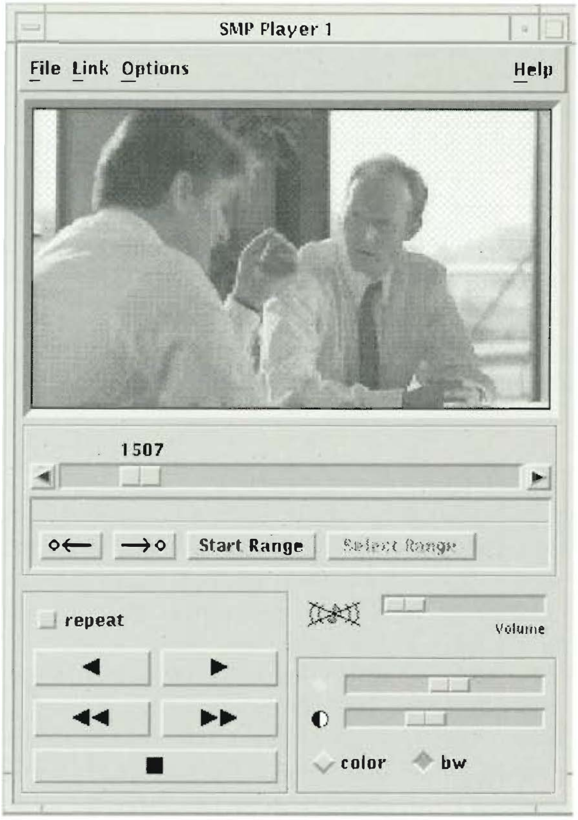


Figure 1 User Controls as Displayed on the Workstation Screen

An example of the latter is the storage of SMP clips directly in a relational database system in which no files exist, such as SQL Multimedia. The video data is stored directly in database records, and the client receives the data through the standard remote database access protocols. At the receiving client, the SMP clip library is used to generate a virtual SMP clip for the application program by substituting a new read function.

The SMP product also contains image converters that translate to and from the popular PBMPLUS family of image formats, allowing import and export to about 70 different image formats, including the Digital Document Interchange Format (DDIF). This allows the use of almost any image format as input for creation of SMP clips.

Historical Background and Requirements

In 1989 Digital's Distributed Multimedia Group experimented briefly with an algorithm called color cell compression (CCC) that had been described in 1986 by Campbell et al.[3] CCC is a coding method that is optimized for rapid decompression of color images in software. We built a demonstrator that rapidly displayed CCC-coded images in a loop to create a motion video effect. The demonstrator then served as our study vehicle to create a usable product for digital video playback.

Performing digital video entirely in software would stress the systems at all levels (I/O, processor, and graphics), so we needed to establish upper bounds for what we could hope to achieve with our desktop systems and workstations.

From the user's perspective, large sizes and high frame rates are desirable. These features need to be balanced with the limitations of real hardware. We modeled the data path through which digital video would have to flow in the system and measured the available resources on the slowest system we would use, a DECstation 2100. This workstation has a 12.5-megahertz (MHz) MIPS R2000 processor and a simple, 8-bit color frame buffer.

By merging this measurement with user feedback concerning the smallest acceptable image size and frame rate, we set our performance goal to play back movies of size 240 by 320 on the slowest DECstation processor with an 8-bit display at 15 frames per second. Smaller viewing sizes are almost invisible on a typical high-resolution workstation screen.

We settled for a frame rate of 15 frames per second. This rate is reasonably smooth: to the human eye, it appears as motion rather than separate images. It can be generated easily from 30-frame source material, such as standard video used in North America and Japan, by taking every other frame. Consequently, on the DECstation 2100 we would have at most

$$\frac{12.5 \times 10^6 \text{ clock cycles/second}}{(320 \times 240 \times 15) \text{ pixels/second}} = 10.85 \text{ clock cycles per pixel}$$

Thus, we must average no more than (approximately) ten machine instructions to decode and render each pixel to the screen.

In order to set our target for compression efficiency, we looked at the volume of data and possible distribution methods. CD-ROM looked promising, and this data rate was also chosen by the Motion Picture Experts Group (MPEG)-1 standard.[4] Hence our coded data rate goal was to maintain a coded data rate for this size and frame rate that would allow playback from a CD-ROM. To achieve this goal, we limited the coded data rate for the video component to 135 to 142 kilobytes per second for video, leaving 8 to 15 kilobytes per second for audio. In addition, we had to limit fluctuations of the coded data rate to allow sensible use of bandwidth reservation protocols for playback over a network without complex buffering schemes.

More interesting were the issues that became apparent when we attempted to use the prototype for real applications. The digital video material had to be usable on a wide range of display types, and due to its large volume, keeping specialized versions for different displays was prohibitive. We would have to adapt the rendition of the coded material to the device-dependent color capabilities of the target display at run time.

Our design center used 8-bit color-mapped displays. These were (and still are) the most common color displays, and the demonstrator was based on them. Integration of the video into applications in a multitasking environment necessitated that computational as well as color resources were available for use by other applications. The system would have to perform cooperative sharing of the scarce color resources on displays with limited color capabilities.

From the perspective of portability, we needed to conform to existing X11 interfaces, without any hidden back doors into the window system. The X Window System affords no direct way of writing into the frame buffer. Rather, the MITSHM extension is used to write an image into a shared memory segment, and then the X server must copy it into the frame buffer. This method would impact our already strained CPU budget for the codec operation. We would need to decompress video in our code and have the X server perform a copy operation of the decompressed video to the screen, again using the main CPU. Quick measurements showed that the copy alone would use approximately 50 percent of the CPU budget for an 8-bit frame buffer, and another 5 to 10 percent would be used by reading the coded data from I/O devices.

With approximately five clock cycles per pixel yet to be rendered, it became clear why none of the standard video algorithms was of any use for such a task. We went back to the original CCC algorithm and started the development of software motion pictures.

Comparison with Other Video Algorithms

Today (early 1993), a number of digital video compression algorithms are in use. All of them are guarded closely as proprietary and therefore closed, and only one algorithm predates the development of SMP. Although we could not build on experiences with these for our work, we believe the internal working on most of them is similar to SMP with some additions.

A popular method for video compression is frame differencing. Rather than each frame being encoded separately, only those parts of the images that have changed relative to a preceding (or future) frame are encoded (together with the information that the other blocks did not change). This method works well for some input material, for example, in video conferences where the camera does not move. The method fails, however, on almost all other video material.

To enable frame differencing on a wider range of input scenes, a method known as motion estimation is used by some algorithms. The encoder for an image sequence performs a search for blocks that have moved between frames and encodes the motion. This search step is computationally very expensive and usually defeats real-time encoding, even for special-purpose hardware.

One of the earliest algorithms was digital video interactive (DVI) from Intel/IBM. It comes in two variations, real-time video (RTV) and production level video (PLV). RTV uses an unknown block encoding scheme and frame differencing. PLV adds motion estimation to this. RTV is comparable to SMP in compression efficiency, computationally more expensive, and much worse in image quality. PLV cannot be done in software and requires special-purpose supercomputers for compression. Compression efficiency of PLV is about twice as good as SMP, and image quality is somewhat better. The more recent INDEO video boards from Intel use RTV.

In 1992 Apple introduced QuickTime, which contains several video compression codecs. The initial RoadPizza (RP) video codec uses simple frame differencing and a block encoding similar to CCC, but without the color quantization step. (This is a guess based on the visual appearance and performance characteristics.) Compression efficiency of RP is three times worse than SMP, and image quality is comparable on 24-bit displays and much worse than SMP on 8-bit displays. Performance is difficult to compare since SMP does not yet run on Macintosh computers.

The newer Compact Video (CV) codec introduced in QuickTime version 1.5 is similar to CCC with frame differencing and has compression efficiency much closer to SMP. Image quality on 8-bit displays is still lower than SMP, and compression times are almost unusable (i.e., long).

The newest entry into the market for software video codecs is the video 1 codec in Microsoft's Video for Windows product. Very little is known about it, but it seems to be close to CCC with frame differencing. Finally, Sun Microsystems has included CCC with frame differencing in their upcoming version of the XIL imaging library.

Three well-known standards for image and video compression have been established by the Joint Photographic Experts Group (JPEG) and the Motion Picture Experts Group (MPEG) committees of the International Organization for Standardization (ISO) and by the Comité Consultatif Internationale de Télégraphique et Téléphonique (CCITT). These standards are computationally too expensive to be performed in software in all but the most powerful workstations today.

The Algorithm

The SMP algorithm is a pixel-based, lossy compression algorithm, designed for minimum CPU loading. It features acceptable image quality, medium compression ratios, and a totally predictable coded data rate. No entropy-based or computationally expensive transform-based coding techniques are used. The downside of this approach is a limited image quality and compression ratio; however, for a wide range of applications, SMP quality is sufficient.

Block Truncation Coding

In 1978, the method referred to as block truncation coding (BTC) was independently reported in the United States by Mitchell, Delp, and Carlton and in Japan by Kishimoto, Mitsuya, and Hoshida.[3,5,6,7]

BTC is a gray-scale image compression technique. The image is first segmented into 4 by 4 blocks. For each block, the 16-pixel average is found and used as a threshold. Each pixel is then assigned to a high or a low group in relation to this threshold. An example of the first stage in the coding process is shown in Figure 2a, in which the sample mean is 101. Each pixel in the block is thus truncated to 1 bit, based on this threshold (see Figure 2b).

12	14	15	23
62	100	201	204
190	195	240	41
20	48	206	45

(a) The average of these 16 pixels is 101.

0	0	0	0
0	0	1	1
1	1	1	0
0	0	1	0

(b) The average of 101 is used as a threshold to segment the block.

Figure 2 Block Truncation Coding of a 4 by 4 Block

For each of the two groups, the average is then calculated again, giving a low average, a, and a high average, b. Mathematically, the first and second statistical moments of the block are preserved. Therefore, for a block of m pixels, with q pixels greater than the sample mean $\bar{x}^2$, and sample variance $\bar{\sigma}^2$, it can be shown that

$$a = \bar{x} - \bar{\sigma}\ \sqrt{q/(m-q)}$$
$$b = \bar{x} + \bar{\sigma}\ \sqrt{(m-q)/q}$$

More intuitively, the bit mask represents the shape of things in the block, and the average luminance and contrast of the block contents are preserved. With this coding method, for blocks of 4 by 4 pixels and 8-bit gray values, a 16-bit mask and two 8-bit values encode the 16 pixels in 32 bits for a rate of 2.0 bits per pixel.

Color Cell Compression

Lema and Mitchell first extended BTC to color by employing a luminance-chrominance space.[8] However, the direction taken by Campbell et al. was computationally faster for decode.[3] In this approach, a luminance value is computed for each pixel. As in the BTC algorithm, the sample mean of the luminance in each 4 by 4 block is used to segment pixels into low and high groups based on luminance values only. The 24-bit color values assigned to the low and high groups are found by independently solving for the 8-bit red, green, and blue values. This allows each block to be represented by a 16-bit mask and two 24-bit color values, for a coding rate of 4 bits per pixel.

The 24-bit values are mapped to a set of 256 8-bit color index values by means of a histogram-based palette selection scheme known as the median cut algorithm.[9] Thus every block can be represented by two 8-bit color indices and the 16-bit mask, yielding 2 bits per pixel; however, each image frame must also send the table of 256 24-bit color values.

Software Motion Pictures Compression

With our goal of 320 by 240 image resolution playback at 15 frames per second, straight CCC coding would have resulted in a data stream of more than 292 kilobytes per second, which is well beyond the capabilities of standard CD-ROM drives. Thus SMP needed to improve the compression ratio of CCC approximately twofold.

Given that we could not apply any of the more expensive compression techniques, we looked for computationally cheap data-reduction techniques. Since most of these techniques negatively impact image quality, we needed a visual test bed to judge the impact of each change.

We computed the images off-line for a short sequence, frame by frame, and then preloaded the images into the workstation memory. The player program then moved the images to the frame buffer in a loop, allowing us to view the results as they would be seen in the final version. The use of this technique provided two advantages. First, we could discover motion artifacts that were invisible in any individual frame. Second, we could judge the covering aspects of motion, which tends to brush over some defects that look objectionable in a still frame.

At first, interframe or frame difference coding looked like a reasonable technique for achieving better compression results without sacrificing image quality, but this was highly dependent on the nature of the input material. Due to the low CPU budget, we could not use any of the more elaborate motion compensation algorithms, so even slight movements in the input video material largely defeated frame differencing. Typically, we achieved only 10 percent better compression with interframe coding, while introducing considerable complexity to the compression and decoding operations. As a result, we dropped interframe coding and made SMP a pure intraframe method, simplifying editing operations and random access to digitized material. At the same time, this opened up use of SMP for still image applications.

To reach our final compression ratio goal of approximately 1 bit per pixel, we settled for a combination of two subsampling techniques. Similar techniques have been independently described by Pins, who conducted an exhaustive search and evaluation of compression techniques.[10] His findings served as a check on our experiments.

Blocks with a low ratio of foreground-to-background luminance (a metric that can be interpreted as contrast) are represented in SMP by a single color and no mask. This reduces the coded representation to a single byte compared to 4 bytes in CCC, which amounts to a fourfold subsampling of such blocks. No chrominance information enters into this decision. It is surprising, but even very marked chrominance differences in foreground/background pairs are readily accepted by the human eye.

With the introduction of a second kind of block, additional encoding information was necessary to distinguish normal (structured) CCC blocks from the subsampled (flat) blocks. In the SMP encoding, this is handled by a bitmap with one bit flagging each block.

Because the adaptive subsampling alone did not yield enough data reduction for our compression goal, we added fixed subsampling for the structured blocks. The horizontal resolution of the structured blocks in SMP is halved relative to CCC by horizontally averaging two neighboring pixels, which reduces the number of bits in the mask from 16 to 8. This reduction leads to blurred vertical edges but looks reasonable for natural video images. Fixed subsampling allowed the encoding of structured blocks with 3 bytes instead of 4 bytes.

We reapplied these ideas to the original grayscale block truncation algorithm. We added a variation to the format that does not use a color look-up table but interprets the foreground and background colors directly as luminance values. Images compressed in this format code gray-scale input material more compactly (there is no need to transmit the leading color look-up table as in CCC); they also do not suffer from the quantization band effects inherent in the color quantization used in the CCC algorithm.

We varied the ratio of flat to structured blocks to effect a trade-off between image quality and compression ratio; however, the range of useful settings is relatively small. If too few structured blocks are allocated, the image essentially is scaled down fourfold, which makes the image look very blocky. If too many structured blocks are allocated, regions of the image that have little detail are encoded with unnecessary overhead. Over the wide range of images we tested, allocating between 30 percent and 50 percent of structured blocks worked best, yielding a bit rate of 0.9 to 1.0 bits per pixel. For color images, the overhead of the color table (768 bytes) must be added.

Decompression

The most challenging part of the design of the SMP system, given the performance requirements, is the decompression step. Efficient rendering techniques of block-truncation coding are well known for certain classes of output devices.[3] SMP improves on the implementations described in the literature by complementing the raw algorithm with efficient, device-independent rendering engines.[3,5,8,10,11] To maximize code efficiency, a separate decompression routine is used for each display situation, rather than using conditionals in a more generic routine. The current implementation can render to 1-, 8-, and 24-bit displays.

Decompression of BTC involves filling 4 by 4 blocks of pixels with two colors under a mask. Because the size and alignment of the blocks is known, a very fast, fully unrolled code sequence can be used. Changes of brightness and contrast of the image can be rapidly adapted to different viewing conditions by manipulating the entries of the colormap of the SMP encoding. Most of the work lies in adaptation of the color content of the decompressed data to the device characteristics of the frame buffer.

For displays with full-color capabilities (24-bit true color), the process is straightforward. The main problem is performing the copy of the decompressed video to the screen. Since 24-bit data is usually allocated in 32-bit words, the amount of data to copy is four times the 8-bit case. Typically, SMP spends 90 percent of the CPU time in the screen copy on 24-bit systems.

The more common and interesting case is to decompress to an 8-bit color representation. Given that SMP is an 8-bit, color-indexed format, it would seem straightforward to download the SMP frame color table to the window system color table and fill the image with the pixel indices directly. This method is impractical for two reasons. First, most window systems (including X11) do not allow reservation of all 256 colors in the hardware color tables. Typically, applications and window managers use a few of the entries for system colors and cursors. Quantizing down to a smaller number of colors (such as 240) could overcome this drawback to a certain degree; however, it would make the SMP-coded material dependent on the device characteristics of a particular window system.

The second and much more problematic aspect is that the SMP frames in a sequence usually have different color tables. Consequently, each frame requires a change of color table that causes a kaleidoscopic effect for the windows of other applications on the screen. In fact, flashing cannot be eliminated within the SMP window itself.

Neither X11 nor other popular window systems such as Microsoft Windows allow reload of the color table and the content of an image at the same time. Therefore, regardless of whether the color table or image contents is modified first, a flashing color effect takes place in the SMP window. It may seem that the update would have to be done in a single screen refresh time as opposed to simultaneously. This is true but irrelevant. Most window systems do not allow for such fine-grain synchronization; and for performance reasons, it was unrealistic to expect to be able to update the image in a single, vertical blanking period.

Alternative suggestions to avoid this problem have been proposed in the literature. One suggestion is to use a single color table for the entire sequence of frames.[10,11] This method is computationally expensive and fails for long sequences and editing operations. Another proposes quantization to less than half of the available colors or partial updates of the color map and use of plane masks.[11] This alternative is not particularly portable between different window systems, and the use of plane masks can have a disastrous impact on performance for some frame-buffer implementations such as the CX adapter in the DECstation product line.

Neither of these methods addresses the issue of monochrome displays or the use of multiple simultaneous SMP movies on a single display. (This effect can be witnessed in Sun Microsystems' recent addition of CCC coding to their XIL library.) To keep device influence out of the compressed material and to enable the use of SMP on a wide range of devices and window systems, a generic decoupling step was added between the colors in the SMP frame and the device colors used for rendition on the screen.

A well-known technique for matching color images to devices with a limited color resolution is dithering. Dithering trades spatial resolution for an apparent increase in color and luminance resolution of the display device. The decrease in spatial resolution is less of an issue for SMP images because of their inherently limited spatial resolution capability. Thus the only challenge was the computational cost of performing dithering in real time.

Fortunately, we found a dithering algorithm that allowed both good quality and high speed.[12] It reduces quantization and mapping to a few table look-up operations, which have a trivial hardware implementation (random access memory) and a reasonable software implementation with a few adds, shifts, and loads.

The general software implementation of the dithering algorithm takes 12 instructions in the MIPS instruction set to map a single pixel to its output representation. For SMP decoding, two different colors at most are in each 4 by 4 block. With this distribution, the cost of dithering is spread over the 16 pixels in each block.

Another optimization used heavily in the 8-bit decoder is to manipulate 4 pixels simultaneously with a single machine instruction. This technique increases performance for decompressing and dithering to 3.2 instructions per pixel in the MIPS instruction set, including all loop overhead, decoding of the encoded data stream, and adjusting contrast and brightness of the image (2.7 instructions per pixel for gray-scale). This efficiency is achieved by careful merging of the decoding, decompression, and dithering phases into a single block of code and avoiding intermediate results written to memory. The cost of the 1-bit and 24-bit decoders is the same or lower (3.2 and 2.9 instructions per pixel, respectively).

Compression

The SMP compressor takes an input image, a desired coded image size, and an output buffer as arguments. It operates in five phases:

- Input scaling (optional)
- Block truncation (luminance)
- Flat block selection
- Color quantization (color SMP only)
- Encoding and output writing

Although the initial scaling is not strictly part of the SMP algorithm, it is necessary for different input sources. Fast scaling is offered as part of both the library and the command-line SMP compressors. Instead of simple subsampling, true averaging is used to ensure maximum input image quality.

The block truncation phase makes two passes through each 4 by 4 block of the input. The first pass calculates the luminance of each individual pixel and sums them to find the average luminance of the entire block. The second pass partitions the pixel pairs into the foreground and background sets and calculates their respective luminance and chrominance averages.

The flat-block-selection phase uses the desired compression ratio to decide how many blocks can be kept as structured blocks and how many need to be converted to flat blocks. The luminance difference of the blocks is calculated, and blocks in the low-contrast range are marked for transition to flat blocks. Because the total average was calculated for each block in the preceding phase, no additional calculations are needed for the conversion of blocks, and the mask is thrown away. Colors are entered into a search structure during this phase.

The color quantization phase uses a median cut algorithm, biased to ensure good coverage of the color contents of the image rather than minimize the overall quantization error. The bias method ensures that small, colored objects are not lost due to large, smoothly shaded areas getting the lion's share of the color allocations. These small objects often are the important features in motion sequences and have a high visibility despite their small size.

The final encoding phase builds the color table and matches the foreground/background colors of the blocks to the best matches in the chosen color table.

The gray-scale compression can be much faster because neither the quantization nor the matching step need be performed. Also, only one-third of the uncompressed video data is usually read in, making gray-scale compression fast enough to enable real-time compression on faster workstations and video-conferencing type applications.

This speed is partly due to the 8-bit restriction in the mask of each structured block. This restriction permits the algorithm to store all intermediate results of the block truncation step in registers on typical reduced instruction set computer (RISC) machines with 32 registers. The entire gray-scale compression algorithm can be done on a MIPS R3000 with 8 machine instructions per input pixel on average, all overhead (except input scaling) included.

Unfortunately, for color processing, SMP compression remains an off-line, non-real-time process, albeit a reasonably fast one at 220 instructions per pixel. A 25-MHz R3000 processor can process more than 40,000 frames in 24 hours (DECstation 5000 Model 200, 320 by 240 at 15 frames per second, TX/PIP as frame grabber), equivalent to 45 minutes of compressed video material per day. The more recent DEC 3000 AXP Model 500 workstation improves this number threefold, so special-purpose hardware for compression is unnecessary even for color SMP.

Portability

A crucial part of the SMP design for portability is the placement of the original SMP codec on the client side of the X Window System. This allows porting and use of SMP on other systems, without being at the mercy of a particular system vendor for integration of the codec into their X server or window system.

This placement is enabled by the efficiency of the SMP decompression engine, which allows many spare cycles for performing the copy of the decompressed, device-dependent video to the window system.

Currently, SMP is offered as a product only on the DECstation family of workstations, but it has been ported to a variety of platforms, including

- DEC AXP workstations running the DEC OSF/1 AXP operating system
- Alpha AXP systems running the OpenVMS operating system
- DECpc AXP personal computers running the Windows NT AXP operating system
- VAX systems running the VMS operating system
- Sun SPARCstation
- IBM RS/6000 system
- HP/PA Precision system
- SCO UNIX/Intel
- Microsoft Windows version 3.1

Generally, porting the SMP system to another platform supporting the X Window System requires the selection of two parameters (host byte order and presence of the MITSHM extension) and then a compilation. The same codec source is used on all the above machines; no assembly language or machine-specific optimizations are used or needed.

The port to Microsoft Windows shows that the same base technology can be used with other window systems, although parts specific to the window system had to be rewritten. The codec code is essentially identical, but the extreme shortage of registers in the 80x86 architecture and the lack of reasonable handling of 32-bit pointers in C language under Windows warrant a rewrite in assembly language on this platform. We do not expect this to be an issue on Windows version 3.2, due to be released later in 1993.

Conclusion

Software motion pictures offers a cost-effective, totally portable way of bringing digital video to the desktop without requiring special investments for add-on hardware. Combined with audio facilities, SMP can be used to bring a complete video playback to most desktop systems. The algorithm and implementation were designed to be used from CD-ROMs as well as network connections. SMP seamlessly integrates with the existing windowing system software. Because of its potentially universal availability, SMP can serve an important function as the lowest common denominator for digital video across multiple platforms.

Acknowledgments

We would like to thank all the people who have contributed to making software motion pictures a reality. Particular thanks go to Paul Tallett for writing the original demonstrator and insisting on the importance of a color version. He also implemented the VMS versions. Thanks also to European External Research for making the initial research and later product transition possible. Last but not least, thanks to Susan Angebranndt and her engineering team for their help and confidence in this work.

References

1. Special Issue on Digital Multimedia Systems, *Communications of the ACM*, vol. 34, no. 4 (April 1991).

2. L. Palmer and R. Palmer, "DECspin: A Networked Desktop Videoconferencing Application," *Digital Technical Journal,* vol. 5, no. 2 (Spring 1993, this issue): 65-76.

3. G. Campbell et al., "Two Bit/Pixel Full Color Encoding," *SIGGRAPH'86 Conference Proceedings,* vol. 20, no. 4 (1986): 215-223.

4. D. LeGall, "MPEG: A Video Compression Standard for Multimedia Applications," *Communications of the ACM,* vol. 34, no. 4 (April 1991): 47-58.

5. O. Mitchell, E. Delp, and S. Carlton, "Block Truncation Coding: A New Approach to Image Compression," *Conference Record, IEEE International Conference Communications,* vol. 1 (June 1978): 12B.1.1-12B.1.4.

6. T. Kishimoto, E. Mitsuya, and K. Hoshida, "A Method of Still Picture Coding by Using Statistical Properties" (in Japanese), *Proceedings of the National Conference of the Institute of Electronics and Communications Engineers of Japan,* no. 974 (March 1978).

7. E. Delp and O. Mitchell, "Image Compression Using Block Truncation Coding," *IEEE Transactions on Communications,* vol. COM-27 (1979): 1335-1342.

8. M. Lema and O. Mitchell, "Absolute Moment Block Truncation Coding and Its Application to Color Images," *IEEE Transactions on Communications,* vol. COM-32, no. 10 (1984): 1148-1157.

9. P. Heckbert, "Color Image Quantization for Frame Buffer Display," *Computer Graphics (AMC SIGGRAPH'82 Conference Proceedings),* vol. 16, no. 3 (1982): 297-307.

10. M. Pins, "Analyse und Auswahl von Algorithmen zur Datenkompression unter besonderer Berücksichtigung von Bildern und Bildfolgen," Ph.D. thesis, University of Karlsrühe, 1990.

11. B. Lamparter and W. Effelsberg, "Digitale Filmübertragung und Darstellung im X-Window-System," Lehrstuhl für Praktische Informatik IV, University of Mannheim, 1991.

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

Davis Yen Pan

Digital Audio Compression

Compared to most digital data types, with the exception of digital video, the data rates associated with uncompressed digital audio are substantial. Digital audio compression enables more efficient storage and transmission of audio data. The many forms of audio compression techniques offer a range of encoder and decoder complexity, compressed audio quality, and differing amounts of data compression. The μ-law transformation and ADPCM coder are simple approaches with low-complexity, low-compression, and medium audio quality algorithms. The MPEG/audio standard is a high-complexity, high-compression, and high audio quality algorithm. These techniques apply to general audio signals and are not specifically tuned for speech signals.

Digital audio compression allows the efficient storage and transmission of audio data. The various audio compression techniques offer different levels of complexity, compressed audio quality, and amount of data compression.

This paper is a survey of techniques used to compress digital audio signals. Its intent is to provide useful information for readers of all levels of experience with digital audio processing. The paper begins with a summary of the basic audio digitization process. The next two sections present detailed descriptions of two relatively simple approaches to audio compression: μ-law and adaptive differential pulse code modulation. In the following section, the paper gives an overview of a third, much more sophisticated, compression audio algorithm from the Motion Picture Experts Group. The topics covered in this section are quite complex and are intended for the reader who is familiar with digital signal processing. The paper concludes with a discussion of software-only real-time implementations.

Digital Audio Data

The digital representation of audio data offers many advantages: high noise immunity, stability, and reproducibility. Audio in digital form also allows the efficient implementation of many audio processing functions (e.g., mixing, filtering, and equalization) through the digital computer.

The conversion from the analog to the digital domain begins by sampling the audio input in regular, discrete intervals of time and quantizing the sampled values into a discrete number of evenly spaced levels. The digital audio data consists of a sequence of binary values representing the number of quantizer levels for each audio sample. The method of representing each sample with an independent code word is called pulse code modulation (PCM). Figure 1 shows the digital audio process.

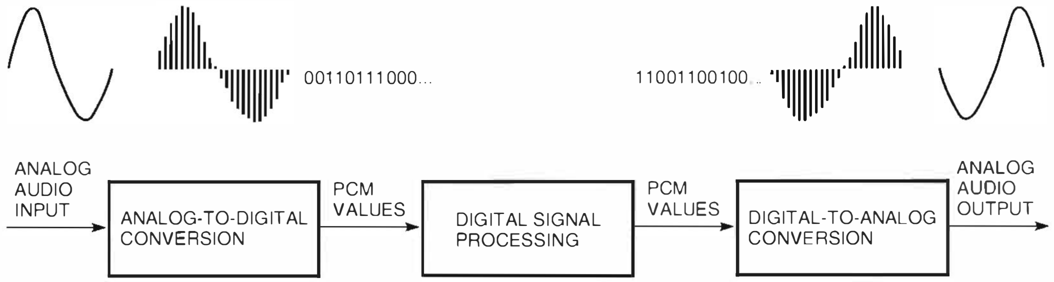


Figure 1 Digital Audio Process

According to the Nyquist theory, a time-sampled signal can faithfully represent signals up to half the sampling rate.[1] Typical sampling rates range from 8 kilohertz (kHz) to 48 kHz. The 8-kHz rate covers a frequency range up to 4 kHz and so covers most of the frequencies produced by the human voice. The 48-kHz rate covers a frequency range up to 24 kHz and more than adequately covers the entire audible frequency range, which for humans typically extends to only 20 kHz. In practice, the frequency range is somewhat less than half the sampling rate because of the practical system limitations.

The number of quantizer levels is typically a power of 2 to make full use of a fixed number of bits per audio sample to represent the quantized values. With uniform quantizer step spacing, each additional bit has the potential of increasing the signal-to-noise ratio, or equivalently the dynamic range, of the quantized amplitude by roughly 6 decibels (dB). The typical number of bits per sample used for digital audio ranges from 8 to 16. The dynamic range capability of these representations thus ranges from 48 to 96 dB, respectively. To put these ranges into perspective, if 0 dB represents the weakest audible sound pressure level, then 25 dB is the minimum noise level in a typical recording studio, 35 dB is the noise level inside a quiet home, and 120 dB is the loudest level before discomfort begins.[2] In terms of audio perception, 1 dB is the minimum audible change in sound pressure level under the best conditions, and doubling the sound pressure level amounts to one perceptual step in loudness.

Compared to most digital data types (digital video excluded), the data rates associated with uncompressed digital audio are substantial. For example, the audio data on a compact disc (2 channels of audio sampled at 44.1 kHz with 16 bits per sample) requires a data rate of about 1.4 megabits per second. There is a clear need for some form of compression to enable the more efficient storage and transmission of this data.

The many forms of audio compression techniques differ in the trade-offs between encoder and decoder complexity, the compressed audio quality, and the amount of data compression. The techniques presented in the following sections of this paper cover the full range from the μ-law, a low-complexity, low-compression, and medium audio quality algorithm, to MPEG/audio, a high-complexity, high-compression, and high audio quality algorithm. These techniques apply to general audio signals and are not specifically tuned for speech signals. This paper does not cover audio compression algorithms designed specifically for speech signals. These algorithms are generally based on a modeling of the vocal tract and do not work well for non-speech audio signals.[3,4] The federal standards 1015 LPC (linear predictive coding) and 1016 CELP (coded excited linear prediction) fall into this category of audio compression.

μ-law Audio Compression

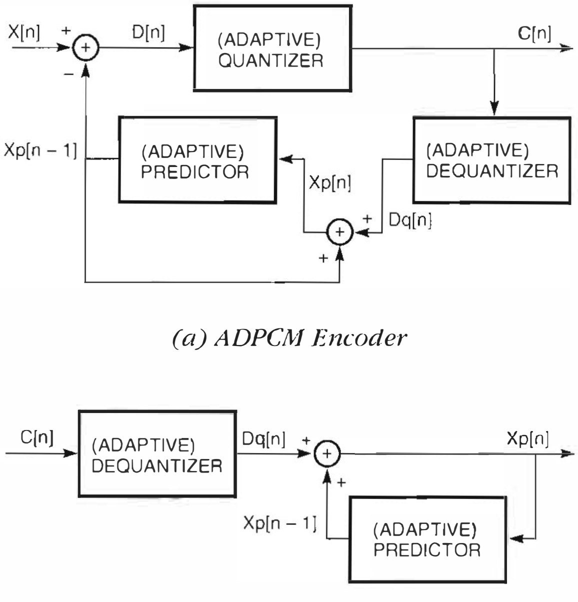
The μ-law transformation is a basic audio compression technique specified by the Comité Consultatif Internationale de Télégraphique et Téléphonique (CCITT) Recommendation G.711.[5] The transformation is essentially logarithmic in nature and allows the 8 bits per sample output codes to cover a dynamic range equivalent to 14 bits of linearly quantized values. This transformation offers a compression ratio of (number of bits per source sample)/8 to 1. Unlike linear quantization, the logarithmic step spacings represent low-amplitude audio samples with greater accuracy than higher-amplitude values. Thus the signal-to-noise ratio of the transformed output is more uniform over the range of amplitudes of the input signal. The μ-law transformation is

$$y = \begin{cases} 255 - \dfrac{127}{\ln(1+\mu)} \times \ln(1+\mu|x|) & \text{for } x \geq 0 \\ 127 - \dfrac{127}{\ln(1+\mu)} \times \ln(1+\mu|x|) & \text{for } x < 0 \end{cases}$$

where $m = 255$, and x is the value of the input signal normalized to have a maximum value of 1. The CCITT Recommendation G.711 also specifies a similar A-law transformation. The μ-law transformation is in common use in North America and Japan for the Integrated Services Digital Network (ISDN) 8-kHz-sampled, voice-grade, digital telephony service, and the A-law transformation is used elsewhere for the ISDN telephony.

Adaptive Differential Pulse Code Modulation

Figure 2 shows a simplified block diagram of an adaptive differential pulse code modulation (ADPCM) coder.[6] For the sake of clarity, the figure omits details such as bit-stream formatting, the possible use of side information, and the adaptation blocks. The ADPCM coder takes advantage of the



(a) ADPCM Encoder

(b) ADPCM Decoder

Figure 2 ADPCM Compression and Decompression

fact that neighboring audio samples are generally similar to each other. Instead of representing each audio sample independently as in PCM, an ADPCM encoder computes the difference between each audio sample and its predicted value and outputs the PCM value of the differential. Note that the ADPCM *encoder* (Figure 2a) uses most of the components of the ADPCM *decoder* (Figure 2b) to compute the predicted values.

The quantizer output is generally only a (signed) representation of the number of quantizer levels. The requantizer reconstructs the value of the quantized sample by multiplying the number of quantizer levels by the quantizer step size and possibly adding an offset of half a step size. Depending on the quantizer implementation, this offset may be necessary to center the requantized value between the quantization thresholds.

The ADPCM coder can adapt to the characteristics of the audio signal by changing the step size of either the quantizer or the predictor, or by changing both. The method of computing the predicted value and the way the predictor and the quantizer adapt to the audio signal vary among different ADPCM coding systems.

Some ADPCM systems require the encoder to provide side information with the differential PCM values. This side information can serve two purposes. First, in some ADPCM schemes the decoder needs the additional information to determine either the predictor or the quantizer step size, or both. Second, the data can provide redundant contextual information to the decoder to enable recovery from errors in the bit stream or to allow random access entry into the coded bit stream.

The following section describes the ADPCM algorithm proposed by the Interactive Multimedia Association (IMA). This algorithm offers a compression factor of (number of bits per source sample)/ 4 to 1. Other ADPCM audio compression schemes include the CCITT Recommendation G.721 (32 kilobits per second compressed data rate) and Recommendation G.723 (24 kilobits per second compressed data rate) standards and the compact disc interactive audio compression algorithm.[7,8]

The IMA ADPCM Algorithm The IMA is a consortium of computer hardware and software vendors cooperating to develop a de facto standard for computer multimedia data. The IMA's goal for its audio compression proposal was to select a public-domain audio compression algorithm able to provide good compressed audio quality with good data compression performance. In addition, the algorithm had to be simple enough to enable software-only, real-time decompression of stereo, 44.1-kHz-sampled, audio signals on a 20-megahertz (MHz) 386-class computer. The selected ADPCM algorithm not only meets these goals, but is also simple enough to enable software-only, real-time encoding on the same computer.

The simplicity of the IMA ADPCM proposal lies in the crudity of its predictor. The predicted value of the audio sample is simply the decoded value of the immediately previous audio sample. Thus the predictor block in Figure 2 is merely a time-delay element whose output is the input delayed by one audio sample interval. Since this predictor is not adaptive, side information is not necessary for the reconstruction of the predictor.

Figure 3 shows a block diagram of the quantization process used by the IMA algorithm. The quantizer outputs four bits representing the signed magnitude of the number of quantizer levels for each input sample.

Adaptation to the audio signal takes place only in the quantizer block. The quantizer adapts the step size based on the current step size and the quantizer output of the immediately previous input.

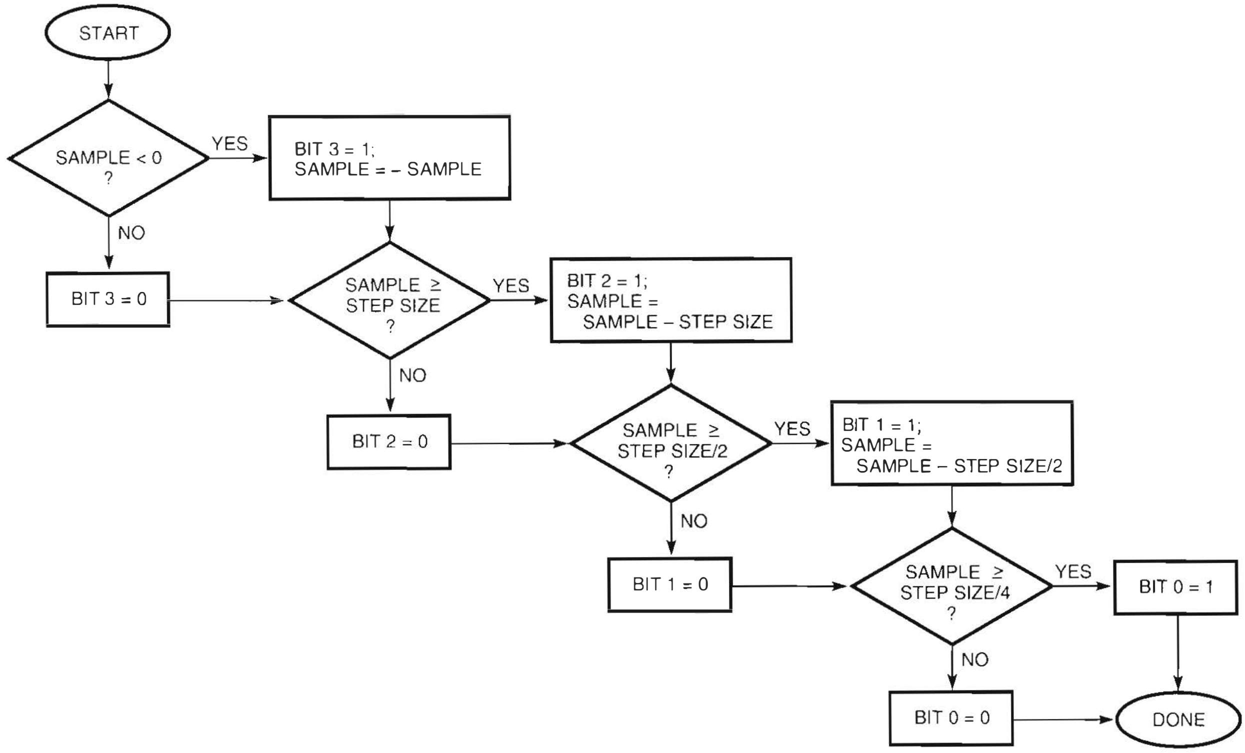


Figure 3 IMA ADPCM Quantization

This adaptation can be done as a sequence of two table lookups. The three bits representing the number of quantizer levels serve as an index into the first table lookup whose output is an index adjustment for the second table lookup. This adjustment is added to a stored index value, and the range-limited result is used as the index to the second table lookup. The summed index value is stored for use in the next iteration of the step-size adaptation. The output of the second table lookup is the new quantizer step size. Note that given a starting value for the index into the second table lookup, the data used for adaptation is completely deducible from the quantizer outputs; side information is not required for the quantizer adaptation. Figure 4 illustrates a block diagram of the step-size adaptation process, and Tables 1 and 2 provide the table lookup contents.

IMA ADPCM: Error Recovery A fortunate side effect of the design of this ADPCM scheme is that decoder errors caused by isolated code word errors or edits, splices, or random access of the compressed bit stream generally do not have a

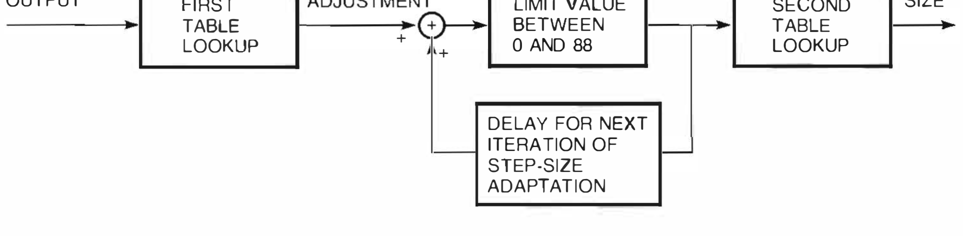

LOWER THREE BITS OF QUANTIZER OUTPUT
FIRST TABLE LOOKUP
INDEX ADJUSTMENT
+
+
+
LIMIT VALUE BETWEEN 0 AND 88
SECOND TABLE LOOKUP
STEP SIZE
DELAY FOR NEXT ITERATION OF STEP-SIZE ADAPTATION

Figure 4 IMA ADPCM Step-size Adaptation

Table 1 First Table Lookup for the IMA ADPCM Quantizer Adaptation

Three Bits Quantized Magnitude	Index Adjustment
000	-1
001	-1
010	-1
011	-1
100	2
101	4
110	6
111	8

disastrous impact on decoder output. This is usually not true for compression schemes that use prediction. Since prediction relies on the correct decoding of previous audio samples, errors in the decoder tend to propagate. The next section explains why the error propagation is generally limited and not disastrous for the IMA algorithm. The decoder reconstructs the audio sample, $Xp[n]$, by adding the previously decoded audio sample, $Xp[n-1]$, to the result of a signed magnitude product of the code word, $C[n]$, and the quantizer step size plus an offset of one-half step size:

$$Xp[n] = Xp[n-1] + \text{step\_size}[n] \times C'[n]$$

where $C'[n]$ = one-half plus a suitable numeric conversion of $C[n]$.

An analysis of the second step-size table lookup reveals that each successive entry is about 1.1 times the previous entry. As long as range limiting of the second table index does not take place, the value for step_size$[n]$ is approximately the product of the previous value, step_size$[n-1]$, and a function of the code word, $F(C[n-1])$:

$$\text{step\_size}[n] = \text{step\_size}[n-1] \times F(C[n-1])$$

The above two equations can be manipulated to express the decoded audio sample, $Xp[n]$, as a

Table 2 Second Table Lookup for the IMA ADPCM Quantizer Adaptation

Index	Step Size	Index	Step Size	Index	Step Size	Index	Step Size
0	7	22	60	44	494	66	4,026
1	8	23	66	45	544	67	4,428
2	9	24	73	46	598	68	4,871
3	10	25	80	47	658	69	5,358
4	11	26	88	48	724	70	5,894
5	12	27	97	49	796	71	6,484
6	13	28	107	50	876	72	7,132
7	14	29	118	51	963	73	7,845
8	16	30	130	52	1,060	74	8,630
9	17	31	143	53	1,166	75	9,493
10	19	32	157	54	1,282	76	10,442
11	21	33	173	55	1,411	77	11,487
12	23	34	190	56	1,552	78	12,635
13	25	35	209	57	1,707	79	13,899
14	28	36	230	58	1,878	80	15,289
15	31	37	253	59	2,066	81	16,818
16	34	38	279	60	2,272	82	18,500
17	37	39	307	61	2,499	83	20,350
18	41	40	337	62	2,749	84	22,358
19	45	41	371	63	3,024	85	24,623
20	50	42	408	64	3,327	86	27,086
21	55	43	449	65	3,660	87	29,794
						88	32,767

function of the step size and the decoded sample value at time, m, and the set of code words between time, m, and n

$$Xp[n] = Xp[m] + \text{step\_size}[m] \times \sum_{i=m+1}^{n} \{\prod_{j=m+1}^{i} F(C[j])\} \times C'[i]$$

Note that the terms in the summation are only a function of the code words from time $m+1$ onward. An error in the code word, $C[q]$, or a random access entry into the bit stream at time q can result in an error in the decoded output, $Xp[q]$, and the quantizer step size, step_size[$q+1$]. The above equation shows that an error in $Xp[m]$ amounts to a constant offset to future values of $Xp[n]$. This offset is inaudible unless the decoded output exceeds its permissible range and is clipped. Clipping results in a momentary audible distortion but also serves to correct partially or fully the offset term. Furthermore, digital high-pass filtering of the decoder output can remove this constant offset term. The above equation also shows that an error in step_size[$m+1$] amounts to an unwanted gain or attenuation of future values of the decoded output $Xp[n]$. The shape of the output wave form is unchanged unless the index to the second step-size table lookup is range limited. Range limiting results in a partial or full correction to the value of the step size.

The nature of the step-size adaptation limits the impact of an error in the step size. Note that an error in step_size[$m+1$] caused by an error in a single code word can be at most a change of $(1.1)^9$, or 7.45 dB in the value of the step size. Note also that any sequence of 88 code words that all have magnitude 3 or less (refer to Table 1) completely corrects the step size to its minimum value. Even at the lowest audio sampling rate typically used, 8 kHz, 88 samples correspond to 11 milliseconds of audio. Thus random access entry or edit points exist whenever 11 milliseconds of low-level signal occur in the audio stream.

MPEG/Audio Compression

The Motion Picture Experts Group (MPEG) audio compression algorithm is an International Organization for Standardization (ISO) standard for high-fidelity audio compression. It is one part of a three-part compression standard. With the other two parts, video and systems, the composite standard addresses the compression of synchronized video and audio at a total bit rate of roughly 1.5 megabits per second.

Like μ-law and ADPCM, the MPEG/audio compression is lossy; however, the MPEG algorithm can achieve transparent, perceptually lossless compression. The MPEG/audio committee conducted extensive subjective listening tests during the development of the standard. The tests showed that even with a 6-to-1 compression ratio (stereo, 16-bit-per-sample audio sampled at 48 kHz compressed to 256 kilobits per second) and under optimal listening conditions, expert listeners were unable to distinguish between coded and original audio clips with statistical significance. Furthermore, these clips were specially chosen because they are difficult to compress. Grewin and Ryden give the details of the setup, procedures, and results of these tests.[9]

The high performance of this compression algorithm is due to the exploitation of auditory masking. This masking is a perceptual weakness of the ear that occurs whenever the presence of a strong audio signal makes a spectral neighborhood of weaker audio signals imperceptible. This noise-masking phenomenon has been observed and corroborated through a variety of psychoacoustic experiments.[10]

Empirical results also show that the ear has a limited frequency selectivity that varies in acuity from less than 100 Hz for the lowest audible frequencies to more than 4 kHz for the highest. Thus the audible spectrum can be partitioned into critical bands that reflect the resolving power of the ear as a function of frequency. Table 3 gives a listing of critical bandwidths.

Because of the ear's limited frequency resolving power, the threshold for noise masking at any given frequency is solely dependent on the signal activity within a critical band of that frequency. Figure 5 illustrates this property. For audio compression, this property can be capitalized by transforming the audio signal into the frequency domain, then dividing the resulting spectrum into subbands that approximate critical bands, and finally quantizing each subband according to the audibility of quantization noise within that band. For optimal compression, each band should be quantized with no more levels than necessary to make the quantization noise inaudible. The following sections present a more detailed description of the MPEG/audio algorithm.

Table 3 Approximate Critical Band Boundaries

Band Number	Frequency (Hz)*	Band Number	Frequency (Hz)*
0	50	14	1,970
1	95	15	2,340
2	140	16	2,720
3	235	17	3,280
4	330	18	3,840
5	420	19	4,690
6	560	20	5,440
7	660	21	6,375
8	800	22	7,690
9	940	23	9,375
10	1,125	24	11,625
11	1,265	25	15,375
12	1,500	26	20,250
13	1,735		

* Frequencies are at the upper end of the band.

MPEG/Audio Encoding and Decoding

Figure 6 shows block diagrams of the MPEG/audio encoder and decoder.[11,12] In this high-level representation, encoding closely parallels the process described above. The input audio stream passes through a filter bank that divides the input into multiple subbands. The input audio stream simultaneously passes through a psychoacoustic model that determines the signal-to-mask ratio of each subband. The bit or noise allocation block uses the signal-to-mask ratios to decide how to apportion the total number of code bits available for the quantization of the subband signals to minimize the audibility of the quantization noise. Finally, the last block takes the representation of the quantized audio samples and formats the data into a decodable bit stream. The decoder simply reverses the formatting, then reconstructs the quantized subband values, and finally transforms the set of subband values into a time-domain audio signal. As specified by the MPEG requirements, ancillary data not necessarily related to the audio stream can be fitted within the coded bit stream.

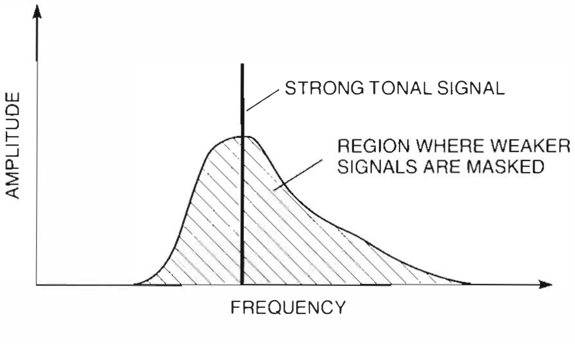
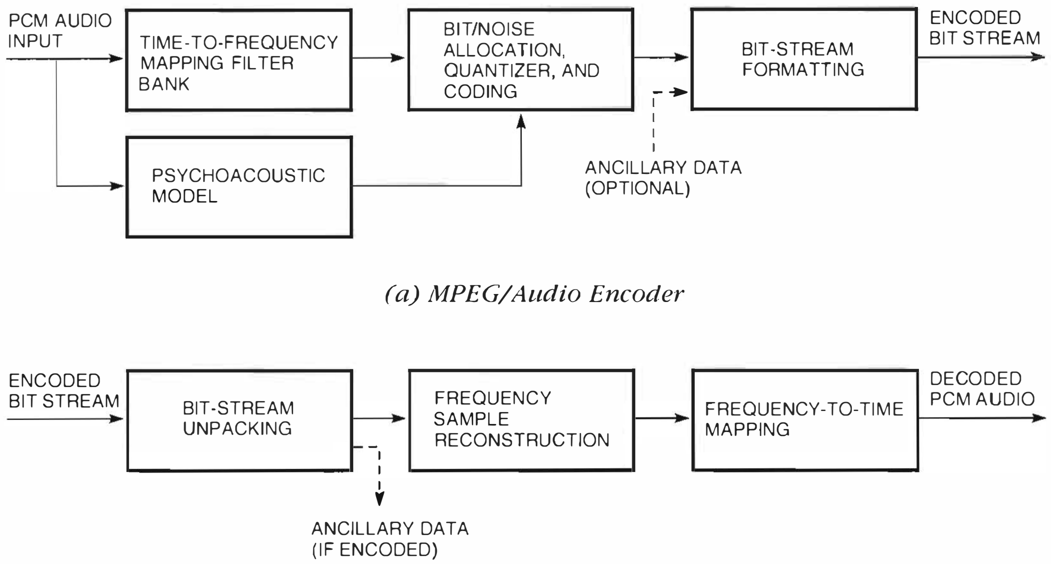


Figure 5 Audio Noise Masking

The MPEG/audio standard has three distinct layers for compression. Layer I forms the most basic algorithm, and Layers II and III are enhancements that use some elements found in Layer I. Each successive layer improves the compression performance but at the cost of greater encoder and decoder complexity.

Layer I The Layer I algorithm uses the basic filter bank found in all layers. This filter bank divides the audio signal into 32 constant-width frequency bands. The filters are relatively simple and provide good time resolution with reasonable frequency resolution relative to the perceptual properties of the human ear. The design is a compromise with three notable concessions. First, the 32 constant-width bands do not accurately reflect the ear's critical bands. Figure 7 illustrates this discrepancy. The bandwidth is too wide for the lower frequencies so the number of quantizer bits cannot be specifically tuned for the noise sensitivity within each critical band. Instead, the included critical band with the greatest noise sensitivity dictates the number of quantization bits required for the entire filter band. Second, the filter bank and its inverse are not lossless transformations. Even without quantization, the inverse transformation would not perfectly recover the original input signal. Fortunately, the error introduced by the filter bank is small and inaudible. Finally, adjacent filter bands have a significant frequency overlap. A signal at a single frequency can affect two adjacent filter bank outputs.

The filter bank provides 32 frequency samples, one sample per band, for every 32 input audio samples. The Layer I algorithm groups together 12 samples from each of the 32 bands. Each group of 12 samples receives a bit allocation and, if the bit allocation is not zero, a scale factor. Coding for stereo redundancy compression is slightly different and is discussed later in this paper. The bit allocation determines the number of bits used to represent each sample. The scale factor is a multiplier that sizes the samples to maximize the resolution of the quantizer. The Layer I encoder formats the


PCM AUDIO INPUT
TIME-TO-FREQUENCY MAPPING FILTER BANK
BIT/NOISE ALLOCATION, QUANTIZER, AND CODING
BIT-STREAM FORMATTING
ENCODED BIT STREAM
PSYCHOACOUSTIC MODEL
ANCILLARY DATA (OPTIONAL)
ENCODED BIT STREAM
BIT-STREAM UNPACKING
FREQUENCY SAMPLE RECONSTRUCTION
FREQUENCY-TO-TIME MAPPING
DECODED PCM AUDIO
ANCILLARY DATA (IF ENCODED)

(a) MPEG/Audio Encoder

(b) MPEG/Audio Decoder

Figure 6 MPEG/Audio Compression and Decompression

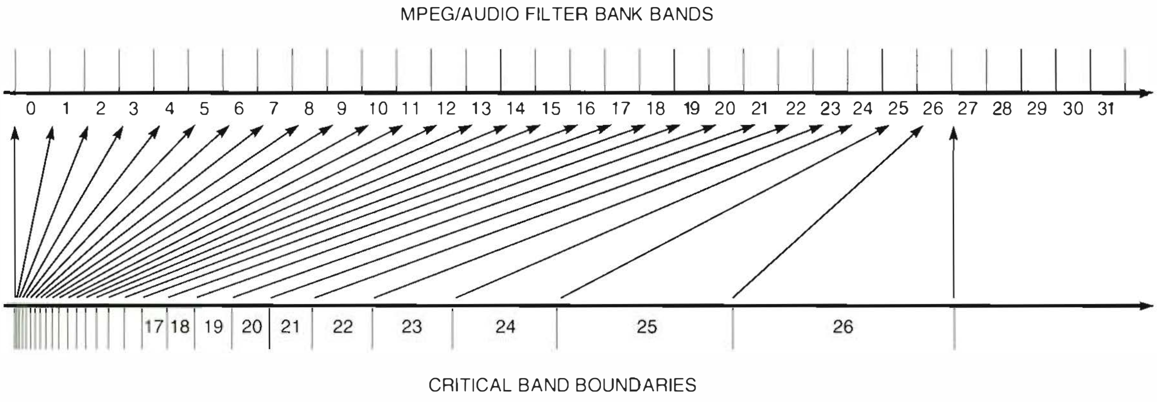


Figure 7 MPEG/Audio Filter Bandwidths versus Critical Bandwidths

32 groups of 12 samples (i.e., 384 samples) into a frame. Besides the audio data, each frame contains a header, an optional cyclic redundancy code (CRC) check word, and possibly ancillary data.

Layer II The Layer II algorithm is a simple enhancement of Layer I. It improves compression performance by coding data in larger groups. The Layer II encoder forms frames of 3 by 12 by 32 = 1,152 samples per audio channel. Whereas Layer I codes data in single groups of 12 samples for each subband, Layer II codes data in 3 groups of 12 samples for each subband. Again discounting stereo redundancy coding, there is one bit allocation and up to three scale factors for each trio of 12 samples. The encoder encodes with a unique scale factor for each group of 12 samples only if necessary to avoid audible distortion. The encoder shares scale factor values between two or all three groups in two other cases: (1) when the values of the scale factors are sufficiently close and (2) when the encoder anticipates that temporal noise masking by the ear will hide the consequent distortion. The Layer II algorithm also improves performance over Layer I by representing the bit allocation, the scale factor values, and the quantized samples with a more efficient code.

Layer III The Layer III algorithm is a much more refined approach.[13,14] Although based on the same filter bank found in Layers I and II, Layer III compensates for some filter bank deficiencies by processing the filter outputs with a modified discrete cosine transform (MDCT). Figure 8 shows a block diagram of the process.

The MDCTs further subdivide the filter bank outputs in frequency to provide better spectral resolution. Because of the inevitable trade-off between time and frequency resolution, Layer III specifies two different MDCT block lengths: a long block of 36 samples or a short block of 12. The short block length improves the time resolution to cope with transients. Note that the short block length is one-third that of a long block; when used, three short blocks replace a single long block. The switch between long and short blocks is not instantaneous. A long block with a specialized long-to-short or short-to-long data window provides the transition mechanism from a long to a short block. Layer III has three blocking modes: two modes where the outputs of the 32 filter banks can all pass through MDCTs with the same block length and a mixed block mode where the 2 lower-frequency bands use long blocks and the 30 upper bands use short blocks.

Other major enhancements over the Layer I and Layer II algorithms include:

- Alias reduction - Layer III specifies a method of processing the MDCT values to remove some redundancy caused by the overlapping bands of the Layer I and Layer II filter bank.
- Nonuniform quantization - The Layer III quantizer raises its input to the 3/4 power before quantization to provide a more consistent signal-to-noise ratio over the range of quantizer values. The requantizer in the MPEG/audio decoder relinearizes the values by raising its output to the 4/3 power.
- Entropy coding of data values - Layer III uses Huffman codes to encode the quantized samples for better data compression.[15]
- Use of a bit reservoir - The design of the Layer III bit stream better fits the variable length nature of the compressed data. As with Layer II, Layer III processes the audio data in frames of 1,152 samples. Unlike Layer II, the coded data representing these samples does not necessarily fit into a fixed-length frame in the code bit stream. The encoder can donate bits to or borrow bits from the reservoir when appropriate.
- Noise allocation instead of bit allocation - The bit allocation process used by Layers I and II only approximates the amount of noise caused by quantization to a given number of bits. The Layer III encoder uses a noise allocation iteration loop. In this loop, the quantizers are varied in an orderly way, and the resulting quantization noise is actually calculated and specifically allocated to each subband.

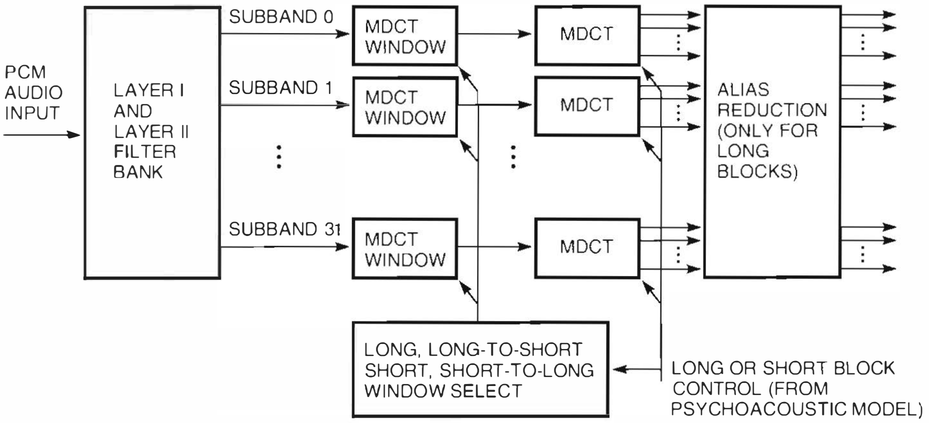

PCM AUDIO INPUT
LAYER I AND LAYER II FILTER BANK
SUBBAND 0
SUBBAND 1
SUBBAND 31
MDCT WINDOW
MDCT WINDOW
MDCT WINDOW
MDCT
MDCT
MDCT
ALIAS REDUCTION (ONLY FOR LONG BLOCKS)
LONG, LONG-TO-SHORT SHORT, SHORT-TO-LONG WINDOW SELECT
LONG OR SHORT BLOCK CONTROL (FROM PSYCHOACOUSTIC MODEL)

Figure 8 MPEG/Audio Layer III Filter Bank Processing, Encoder Side

The Psychoacoustic Model

The psychoacoustic model is the key component of the MPEG encoder that enables its high performance.[16,17,18,19] The job of the psychoacoustic model is to analyze the input audio signal and determine where in the spectrum quantization noise will be masked and to what extent. The encoder uses this information to decide how best to represent the input audio signal with its limited number of code bits. The MPEG/audio standard provides two example implementations of the psychoacoustic model. Below is a general outline of the basic steps involved in the psychoacoustic calculations for either model.

- Time align audio data - The psychoacoustic model must account for both the delay of the audio data through the filter bank and a data offset so that the relevant data is centered within its analysis window. For example, when using psychoacoustic model two for Layer I, the delay through the filter bank is 256 samples, and the offset required to center the 384 samples of a Layer I frame in the 512-point psychoacoustic analysis window is $(512 - 384)/2 = 64$ points. The net offset is 320 points to time align the psychoacoustic model data with the filter bank outputs.

- Convert audio to spectral domain - The psychoacoustic model uses a time-to-frequency mapping such as a 512- or 1,024-point Fourier transform. A standard Hann weighting, applied to audio data before Fourier transformation, conditions the data to reduce the edge effects of the transform window. The model uses this separate and independent mapping instead of the filter bank outputs because it needs finer frequency resolution to calculate the masking thresholds.

- Partition spectral values into critical bands - To simplify the psychoacoustic calculations, the model groups the frequency values into perceptual quanta.

- Incorporate threshold in quiet - The model includes an empirically determined absolute masking threshold. This threshold is the lower bound for noise masking and is determined in the absence of masking signals.

- Separate into tonal and nontonal components - The model must identify and separate the tonal and noiselike components of the audio signal because the noise-masking characteristics of the two types of signal are different.

- Apply spreading function - The model determines the noise-masking thresholds by applying an empirically determined masking or spreading function to the signal components.

- Find the minimum masking threshold for each subband - The psychoacoustic model calculates the masking thresholds with a higher-frequency resolution than provided by the filter banks. Where the filter band is wide relative to the critical band (at the lower end of the spectrum), the model selects the minimum of the masking thresholds covered by the filter band. Where the filter band is narrow relative to the critical band, the model uses the average of the masking thresholds covered by the filter band.

- Calculate signal-to-mask ratio - The psychoacoustic model takes the minimum masking threshold and computes the signal-to-mask ratio; it then passes this value to the bit (or noise) allocation section of the encoder.

Stereo Redundancy Coding

The MPEG/audio compression algorithm supports two types of stereo redundancy coding: intensity stereo coding and middle/side (MS) stereo coding. Both forms of redundancy coding exploit another perceptual weakness of the ear. Psychoacoustic results show that, within the critical bands covering frequencies above approximately 2 kHz, the ear bases its perception of stereo imaging more on the temporal envelope of the audio signal than its temporal fine structure. All layers support intensity stereo coding. Layer III also supports MS stereo coding.

In intensity stereo mode, the encoder codes some upper-frequency filter bank outputs with a single summed signal rather than send independent codes for left and right channels for each of the 32 filter bank outputs. The intensity stereo decoder reconstructs the left and right channels based only on independent left- and right-channel scale factors. With intensity stereo coding, the spectral shape of the left and right channels is the same within each intensity-coded filter bank signal, but the magnitude is different.

The MS stereo mode encodes the signals for left and right channels in certain frequency ranges as middle (sum of left and right) and side (difference of left and right) channels. In this mode, the encoder uses specially tuned techniques to further compress side-channel signal.

Real-time Software Implementations

The software-only implementations of the μ-law and ADPCM algorithms can easily run in real time. A single table lookup can do μ-law compression or decompression. A software-only implementation of the IMA ADPCM algorithm can process stereo, 44.1-kHz-sampled audio in real time on a 20-MHz 386-class computer. The challenge lies in developing a real-time software implementation of the MPEG/audio algorithm. The MPEG standards document does not offer many clues in this respect. There are much more efficient ways to compute the calculations required by the encoding and decoding processes than the procedures outlined by the standard. As an example, the following section details how the number of multiplies and additions used in a certain calculation can be reduced by a factor of 12.

Figure 9 shows a flow chart for the analysis subband filter used by the MPEG/audio encoder. Most of the computational load is due to the second-from-last block. This block contains the following matrix multiply:

$$S(i) = \sum_{k=0}^{63} Y(k) \times \cos\left[\frac{(2 \times i+1) \times (k-16) \times \Pi}{64}\right]$$
$$\text{for } i = 0 \ldots 31.$$

Using the above equation, each of the 31 values of $S(i)$ requires 63 adds and 64 multiplies. To optimize this calculation, note that the $M(i,k)$ coefficients are similar to the coefficients used by a 32-point, un-normalized inverse discrete cosine transform (DCT) given by

$$f(i) = \sum_{k=0}^{31} F(k) \times \cos\left[\frac{(2 \times i+1) \times k \times \Pi}{64}\right]$$
$$\text{for } i = 0 \ldots 31.$$

Indeed, $S(i)$ is identical to $f(i)$ if $F(k)$ is computed as follows

$$\begin{aligned} F(k) &= Y(16) \text{ for } k = 0; \\ &= Y(k+16) + Y(16-k) \text{ for } k = 1 \ldots 16; \\ &= Y(k+16) - Y(80-k) \text{ for } k = 17 \ldots 31. \end{aligned}$$

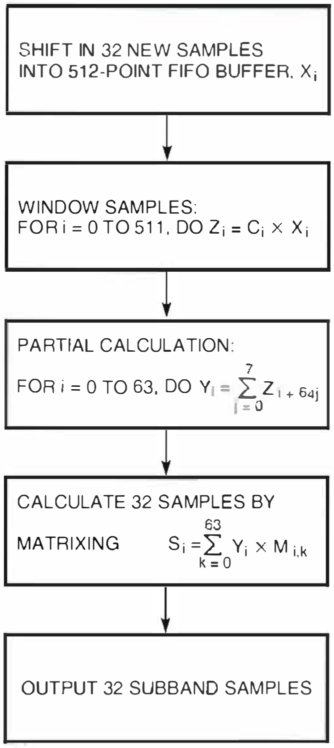

SHIFT IN 32 NEW SAMPLES INTO 512-POINT FIFO BUFFER, X_i
WINDOW SAMPLES: FOR i = 0 TO 511, DO $Z_i = C_i \times X_i$
PARTIAL CALCULATION: FOR i = 0 TO 63, DO $Y_i = \sum_{j=0}^{7} Z_{i+64j}$
CALCULATE 32 SAMPLES BY MATRIXING $S_i = \sum_{k=0}^{63} Y_i \times M_{i,k}$
OUTPUT 32 SUBBAND SAMPLES

Figure 9 Flow Diagram of the MPEG/Audio Encoder Filter Bank

Thus with the almost negligible overhead of computing the $F(k)$ values, a twofold reduction in multiplies and additions comes from halving the range that k varies. Another reduction in multiplies and additions of more than sixfold comes from using one of many possible fast algorithms for the computation of the inverse DCT.[20,21,22] There is a similar optimization applicable to the 64 by 32 matrix multiply found within the decoder's subband filter bank.

Many other optimizations are possible for both MPEG/audio encoder and decoder. Such optimizations enable a software-only version of the MPEG/audio Layer I or Layer II decoder (written in the C programming language) to obtain real-time performance for the decoding of high-fidelity monophonic audio data on a DECstation 5000 Model 200. This workstation uses a 25-MHz R3000 MIPS CPU and has 128 kilobytes of external instruction and data cache. With this optimized software, the MPEG/audio Layer II algorithm requires an average of 13.7 seconds of CPU time (12.8 seconds of user time and 0.9 seconds of system time) to decode 7.47

seconds of a *stereo* audio signal sampled at 48 kHz with 16 bits per sample.

Although real-time MPEG/audio decoding of stereo audio is not possible on the DECstation 5000, such decoding is possible on Digital's workstations equipped with the 150-MHz DECchip 21064 CPU (Alpha AXP architecture) and 512 kilobytes of external instruction and data cache. Indeed, when this same code (i.e., without CPU-specific optimization) is compiled and run on a DEC 3000 AXP Model 500 workstation, the MPEG/audio Layer II algorithm requires an average of 4.2 seconds (3.9 seconds of user time and 0.3 seconds of system time) to decode the same 7.47-second audio sequence.

Summary

Techniques to compress general digital audio signals include μ-law and adaptive differential pulse code modulation. These simple approaches apply low-complexity, low-compression, and medium audio quality algorithms to audio signals. A third technique, the MPEG/audio compression algorithm, is an ISO standard for high-fidelity audio compression. The MPEG/audio standard has three layers of successive complexity for improved compression performance.

References

1. A. Oppenheim and R. Schafer, *Discrete Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall, 1989): 80-87.

2. K. Pohlman, *Principles of Digital Audio* (Indianapolis, IN: Howard W. Sams and Co., 1989).

3. J. Flanagan, *Speech Analysis Synthesis and Perception* (New York: Springer-Verlag, 1972).

4. B. Atal, "Predictive Coding of Speech at Low Rates," IEEE Transactions on Communications, vol. COM-30, no. 4 (April 1982).

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

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

7. M. Nishiguchi, K. Akagiri, and T. Suzuki, "A New Audio Bit Rate Reduction System for the CD-I Format," Preprint 2375, *81st Audio Engineering Society Convention,* Los Angeles (1986).

8. Y. Takahashi, H. Yazawa, K. Yamamoto, and T. Anazawa, "Study and Evaluation of a New Method of ADPCM Encoding," Preprint 2813, *86th Audio Engineering Society Convention,* Hamburg (1989).

9. C. Grewin and T. Ryden, "Subjective Assessments on Low Bit-rate Audio Codecs," *Proceedings of the Tenth International Audio Engineering Society Conference,* London (1991): 91-102.

10. J. Tobias, *Foundations of Modern Auditory Theory* (New York and London: Academic Press, 1970): 159-202.

11. K. Brandenburg and G. Stoll, "The ISO/MPEG-Audio Codec: A Generic Standard for Coding of High Quality Digital Audio," Preprint 3336, *92nd Audio Engineering Society Convention,* Vienna (1992).

12. K. Brandenburg and J. Herre, "Digital Audio Compression for Professional Applications," Preprint 3330, *92nd Audio Engineering Society Convention,* Vienna (1992).

13. K. Brandenburg and J. D. Johnston, "Second Generation Perceptual Audio Coding: The Hybrid Coder," Preprint 2937, *88th Audio Engineering Society Convention,* Montreaux (1990).

14. K. Brandenburg, J. Herre, J. D. Johnston, Y. Mahieux, and E. Schroeder, "ASPEC: Adaptive Spectral Perceptual Entropy Coding of High Quality Music Signals," Preprint 3011, *90th Audio Engineering Society Convention,* Paris (1991).

15. D. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE,* vol. 40 (1962): 1098-1101.

16. J. D. Johnston, "Estimation of Perceptual Entropy Using Noise Masking Criteria," *Proceedings of the 1988 IEEE International Conference on Acoustics, Speech, and Signal Processing* (1988): 2524-2527.

17. J. D. Johnston, "Transform Coding of Audio Signals Using Perceptual Noise Criteria," *IEEE Journal on Selected Areas in Communications,* vol. 6 (February 1988): 314-323.

18. K. Brandenburg, "OCF—A New Coding Algorithm for High Quality Sound Signals," *Proceedings of the 1987 IEEE ICASSP* (1987): 141-144.

19. D. Wiese and G. Stoll, "Bitrate Reduction of High Quality Audio Signals by Modeling the Ear's Masking Thresholds," Preprint 2970, *89th Audio Engineering Society Convention,* Los Angeles (1990).

20. J. Ward and B. Stanier, "Fast Discrete Cosine Transform Algorithm for Systolic Arrays," *Electronics Letters,* vol. 19, no. 2 (January 1983).

21. J. Makhoul, "A Fast Cosine Transform in One and Two Dimensions," *IEEE Transactions on Acoustics, Speech, and Signal Processing,* vol. ASSP-28, no. 1 (February 1980).

22. W.-H. Chen, C. H. Smith, and S. Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform," *IEEE Transactions on Communications,* vol. COM-25 no. 9 (September 1977).

Jan B. te Kiefte
Robert Hasenaar
Joop W. Mevius
Theo H. van Hunnik

The Megadoc Image Document Management System

Megadoc image document management solutions are the result of a systems engineering effort that combined several disciplines, ranging from optical disk hardware to an image application framework. Although each of the component technologies may be fairly mature, combining them into easy-to-customize solutions presented a significant systems engineering challenge. The resulting application framework allows the configuration of customized solutions with low systems integration cost and short time to deployment.

Electronic Document Management

In most organizations, paper is the main medium for information sharing. Paper is not only a communication medium but in many cases also the carrier of an organization's vital information assets. Whereas the recording of information in document format is done largely with help of electronic equipment, sharing and distribution of that information is in many cases still done on paper. Large-scale, paper-based operations have limited options for tracking the progress of work.

The computer industry thus has two opportunities:

1. Capture paper documents in electronic image format (if using paper is a requirement)

2. Provide better tools for sharing and distribution among work groups (if the use of paper can be avoided)

Organizations that use electronic imaging, as compared to handling paper, can better track work in progress. Productivity increases (no time is wasted in searching) and the quality of service improves (response times are shorter and no information is lost) when vital information is represented and tracked electronically.

Imaging is not a new technology (see Table 1). Moreover, this paper does not document new base technology. Instead, we describe the key components of an image document management system in the context of a systems engineering effort. This effort resulted in a product set that allows the configuration of customized solutions.

Those who first adopted the use of image technology have had to go through a long learning curve—a computer with a scanner and an optical disk does not fully address the issues of a large-scale, paper-based operation. Early adopters of electronic imaging experienced a challenge in defining the right electronic document indexing scheme for their applications. Even though the technology is now mature, the introduction of a document imaging system frequently leads to some form of business process reengineering to exploit the new options of electronic document management. The Megadoc image document management system allows the configuration of customer-specific solutions through its building-block architecture and its built-in customization options.

The Megadoc system presented in this paper is based on approximately 10 years of experience with base technology, customer projects, and everything in between. In those years, Megadoc image document management has matured from the technology delight of optical recording to an application framework for image document management. This framework consists of hardware and software components arranged in various architectural layers: the base system, the optical file server, the storage manager, and the image application framework.

The base system consists of PC-based workstations, running the Microsoft Windows operating system, connected to servers for storage management and to database services for document indexing. Specific peripherals include image scanners, image printers, optional full-screen displays, and optional write once, read many (WORM) disks.

The optical file server abstracts from the differences between optical WORM disks and provides

Table 1 History of Image Document Management

1975	Philips Research combines a 12-inch (30.48-centimeter) videodisk for analog storage of facsimile documents and high-resolution video monitors with a minicomputer for indexing in an experimental image management system.
1979	Philips' image management system switches to digital technology through the availability of WORM disks and random-access memory (RAM) chips (for refreshing a full-page video monitor).
1983	At the Hannover Fair (Hannover, Germany), Philips shows Megadoc, an image document management system with WORM disks containing compressed document images. Dedicated image document management solutions are introduced.
1988	Image document management transitions from dedicated image display technology as part of a proprietary computer architecture to an open systems platform with PC-based image workstations.
1993	The image becomes just another document format that is used next to text-coded electronic documents.

the many hundreds of gigabytes (GB) of storage required in large-scale image document management systems.

The storage manager provides storage and retrieval functions for the contents of documents. Document contents are stored in "containers," i.e., large, one-dimensional storage areas that can span multiple optical disk volumes.

The Megadoc image application framework contains three sublayers:

1. Image-related software libraries for scanning, viewing, and printing
2. Application templates
3. A standard folder management application that provides, with some tailoring by the end-user organization, an "out-of-the-box" image document management solution

The optical file server and the storage manager store images in any type of document format. However, to meet customer requirements with respect to longevity of the documents, images should be stored in compressed format according to the Comité Consultatif Internationale de Télégraphique et Téléphonique (CCITT) Group 4 standard.

In addition to image document management solutions, Megadoc components are used to "image enable" existing data processing applications. In many cases, a data processing application uses some means of identification for an application object (e.g., an order or an invoice). This identification relates to a paper document. Megadoc reuses the application's identification as the key to the image version of that document. Application programming interfaces (APIs) for terminal emulation packages that are running the original application in a window on the Megadoc image PC workstations allow integration with the unchanged application.

The following sections describe the optical file server, the storage manager, and the image application framework.

Megadoc Optical File Server

The Megadoc optical file server (OFS) software provides a UNIX file system interface for WORM disks. The OFS automatically loads and unloads these WORM volumes by jukebox robotics in a completely transparent way. Thus, from an API perspective, OFS implements a UNIX file system with a large on-line file system storage capacity. Currently, up to 800 GB can be reached with a single jukebox.

We implemented the OFS in three layers, as shown Figure 1:

1. The optical disk filer (ODF) layer, which enables storing data on write-once devices and providing a UNIX file system interface.
2. The volume manager (VM), which loads and unloads volumes to and from drives in the jukeboxes and communicates with the system operator for handling off-line volumes.
3. The device layer, which provides device-level access to the WORM drives and to the jukebox hardware. This layer is not discussed further in this paper.

Optical Disk Filer

When we started to design the ODF, the chief prerequisite was that it should adhere to the UNIX file system interface for applications. The obvious

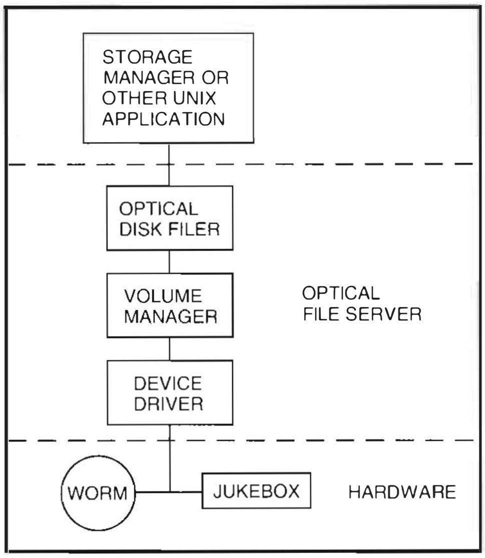


Figure 1 The Three Software Layers of the Optical File Server

benefit was that the designers would not have to write their own utilities to, for example, copy data, create new files, and make new directories. All UNIX utilities would work as well on WORM devices as on any other file system.

Current UNIX implementations provide two kernel interfaces for integrating a new file system type into the kernel: the file system switch (FSS), in UNIX versions based on the System V Release 3; and the virtual file system (VFS), in UNIX implementations like the System V Release 4, SunOS, and OSF/1 operating systems. We introduced the optical disk filer in the FSS and later ported it to the VFS.

The key challenge for the design of a file system for write-once devices is to allow updates without causing an "avalanche" of updates. Note that any update to a sector on a WORM device forces a rewrite of the full sector at another location. If pointers to an updated sector exist on the WORM device, sectors that contain those pointers have to be rewritten, also. For example, if a file system implementation is chosen where the list of data blocks for a file, or just the sector location of such a list, is part of the file's directory information, any update to that file would cause a rewrite of the directory sector and the sectors for the parent directories, all the way up to the root directory.

A second issue to be addressed for removable optical disks is performance. Access time for on-line disks is at least eight times slower than for current magnetic disks. (The average seek time for a WORM device is 100 milliseconds; rotational delay is about 35 milliseconds.) Fetching a disk from a jukebox storage slot, loading it, and waiting for spin-up takes between 8 and 15 seconds, depending on the type of jukebox.

Caching solves both issues. We decided that the usual in-memory cache would not be sufficient for the huge amounts of WORM data, and therefore, we use partitions of magnetic disks for caching.

ODF WORM Layout To avoid duplicating previous efforts, we used classical UNIX file systems as a guideline for the definition of ODF's WORM layout. However, we had to add some indirect pointer mechanisms to avoid update avalanches. Each file system is mapped onto a single WORM partition. These partitions are written sequentially, reducing the free block administration to maintaining a current write point.

The ODF reuses many notions from UNIX file systems, such as i-nodes, superblock, and the functional contents of directory entries.[1] Applying these UNIX notions to the optical file system resulted in the following ODF characteristics:

- The superblock contains all global data for a file system.
- Each i-node contains the block list and all the attributes of a file except the file's name.
- An i-node number identifies each i-node.
- A directory is a special type of file.
- Entries in a directory map names to i-node numbers.

A new notion in the ODF, as compared to UNIX file systems, is the administration file (admin file). One such file exists for each file system. The file is sequential, and its contents are similar to the first disk blocks in classical UNIX file systems: the first extent contains the superblock, and all other extents form a constantly growing array of i-nodes; the i-node's number is the index of the i-node in the file's i-node array. An important difference between UNIX file systems and the ODF is that the 2-kilobyte (kB), fixed-size extents of the ODF admin file are scattered over the WORM device, instead of being stored as a sequential array of disk blocks, as in UNIX systems. As a result, any update to an i-node, as a consequence of a file update, causes the invalidation of at most one admin file extent. Since the logical index in the admin file of this i-node, i.e., the i-node number, does not change, the parent directories do not have to be updated.

However, this scheme needs an additional indirect pointer mechanism: a list of block numbers representing the location of the admin file extents. The ODF stores this list in the admin file's i-node (aino). The aino is a sequential file that contains slightly more than block numbers and is a sequence of contiguous blocks on the WORM disk that contain the same information. Hence, an update to an admin file extent always invalidates the entire aino on the WORM device, which makes the aino a more desirable candidate for caching than the admin file extents.

The following example, shown in Figure 2, illustrates the steps involved in reading logical block *N* from the file with i-node number *I*:

1. Read the aino to obtain the block number of *I*'s admin file extent.
2. Read the admin file extent to get file *I*, which is used to translate the logical block number *N* into the physical block number *I(N)*.
3. Read physical block *I(N)*.

If the file system is in a consolidated state, i.e., all data on the WORM disk is current, the aino and the superblock are the last pieces of information written to the WORM device, directly before the current write point. Blocks written prior to the aino and the superblock contain mainly user data but also an occasional admin file extent, fully interleaved. Figure 3 shows the WORM layout. Since ODF requires the first admin file extent and the complete aino to be in the cache, introducing a disk with consolidated file systems to another system requires searching the current write point, reading the superblock, determining the aino length from the superblock, and finally reading the aino itself. Searching the current write point is a fairly fast operation implemented through binary search and hardware support, which allow the ODF to distinguish between used and unused data blocks of 1K bytes.

ODF Caching Caching in the ODF is file oriented. We suggest a magnetic cache size of approximately 5 percent of the optical disk space. If data from a file on a WORM disk is read, the ODF creates a cache file and copies a contiguous segment of file data from the WORM disk (64 kB in size, or less in the case of a small file) to the correct offset in the cache file. The cache file is the basis for all I/O operations until removed by the ODF, after having rewritten all dirty segments (i.e., updated or changed segments) back to the WORM device. The ODF provides special system calls (through the UNIX fcntl(2) interface) to flush asynchronously dirty file segments to the WORM device and to remove a file's cache file. The flusher daemon monitors high and low watermarks for dirty cache contents. The daemon flushes dirty data to the optical disks. The flusher daemon flushes data in a sequence that minimizes the number of WORM volume movements in a jukebox. The ODF deletes clean data (i.e., data already present on the optical disk) on a least-recently-used basis.

The admin file has its own cache file. The minimum amount of admin file data to be cached is the superblock. The ODF gradually caches the other admin file extents, which contain the i-nodes, while the file system is in use. The ODF writes i-node updates to the WORM device as soon as all i-nodes in the same admin file extent have their dirty file data written to the WORM device. The aino has its own cache file, also, and is always completely cached. If all file data and i-nodes have been written to the WORM device, the file system can be consolidated by a special utility that writes aino and superblock

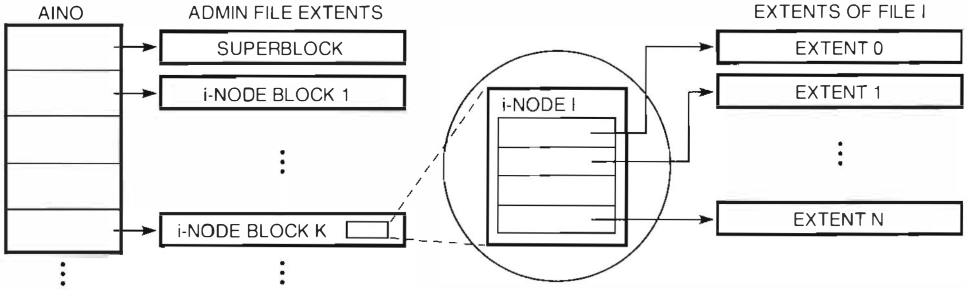


Figure 2 Steps Involved in Getting from the Aino to Extent N of File I

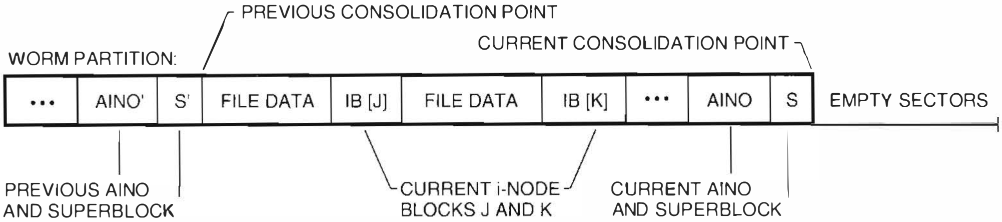

PREVIOUS CONSOLIDATION POINT
CURRENT CONSOLIDATION POINT
WORM PARTITION:
...
AINO'
S'
FILE DATA
IB [J]
FILE DATA
IB [K]
...
AINO
S
EMPTY SECTORS
PREVIOUS AINO AND SUPERBLOCK
CURRENT i-NODE BLOCKS J AND K
CURRENT AINO AND SUPERBLOCK

Figure 3 WORM Layout for a Consolidated ODF File System

to the WORM device, hence creating a consolidation point.

For reasons of modularity and ease of implementation, we chose the UNIX standard magnetic disk file system implementation to perform the caching. An alternative would have been to use a magnetic disk cache with an optimized, ODF-specific structure. We opted for a small amount of overhead, which would allow us to add a faster file system, should one become available. Our performance measurements showed a loss of less than 10 percent in performance as compared to that of an ODF-specific solution. The cache file systems on magnetic disk can be accessed only through the ODF kernel component. Thus, in an active OFS system, no application can access and, therefore, possibly corrupt the cached data.

Volume Manager

In addition to hiding the WORM nature of the underlying physical devices, the OFS transparently moves volumes between drives and storage slots in jukeboxes that contain many volumes ("platters"). The VM performs this function.

The essential characteristic of the volume management layer is its simple functionality, which is best described as a "volume faulting device." The interface to the VM consists of volume device entries, each of which gives access to a specific WORM volume in the system. For example, the volume device entry /dev/WORM_A gives access to the WORM volume WORM_A. This volume device entry has exactly the same interface as the usual device entry such as /dev/worm, which gives access to a specific WORM drive in the system, or rather to any volume that happens to be on that drive at that moment. Any access to a volume device, e.g., /dev/WORM_A, either passes directly to the drive on which the volume (WORM_A) is loaded, or results in a volume fault. This last situation occurs when the volume is in a jukebox slot and not in a directly accessible drive. Note that since /dev/WORM_A has the same interface as /dev/worm, the OFS could function without the VM layer in any system that contains only one worm drive and one volume that is never removed from that drive. However, since this configuration is not a realistic option, the OFS includes the VM layer.

The internal architecture of the VM is more complicated than its functionality might indicate. The VM consists of a relatively small kernel component and several server processes, as illustrated in Figure 4. The kernel component is a pseudo-device driver layer that receives requests for the volume devices, e.g., /dev/WORM_A, and translates these requests into physical device driver (/dev/worm) requests using a table that contains the locations of loaded volumes. If the location of a volume can be found in the table, the I/O request is directly passed on to the physical device. Otherwise, a message is prepared for the central VM server process, and the volume server and the requesting application are put in a waiting state.

The volume server uses a file to translate volume device numbers into volume names and locations. It communicates with two other types of VM server processes: jukebox servers and drive servers. The jukebox servers take care of all movements in their jukebox. Drive servers spin up and spin down their drive only on request from the volume server.

Storage Manager

The storage manager implements containers, as mentioned in the Electronic Document Management section. Large-scale document management uses indexing of multiple storage and retrieval attributes, typically with the help of a relational database. Once the contents of a document are identified through a database query on its attributes, a single pointer to the contents is sufficient.

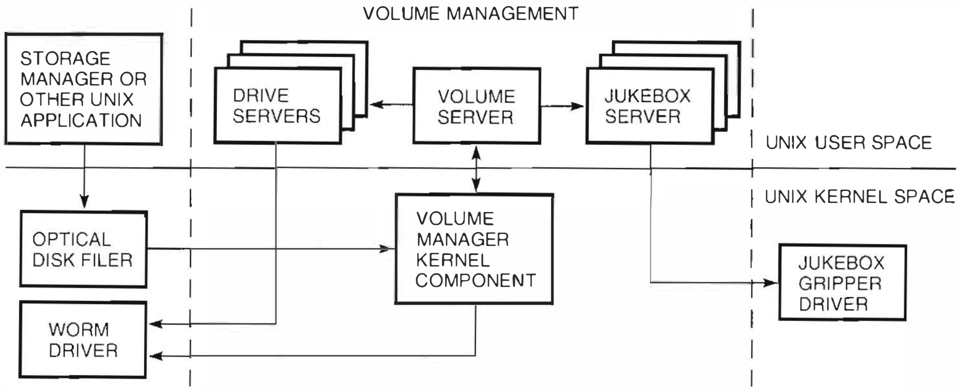


Figure 4 Global Architecture Showing the VM Component

Also, there is little need for a hierarchically structured file system. Containers provide large, flat structures where the contents of a document are uniquely defined by the container identification and a unique identification within the container. The document's contents identification is translated by the storage manager in a path to a directory where one or more contents files can be written. For multipage image documents, the Megadoc system stores each page as a separate image file in a directory reserved for the document. This scheme guarantees locality of reference, avoiding unnatural delays while browsing a multipage image document.

A container consists of a sequence of file systems, typically spanning multiple volumes. Due to the nature of the OFS, no distinction has to be made between WORM disk file systems and magnetic disk file systems. The storage manager fills containers sequentially, up to a configurable threshold for each file system, allowing some degree of local updates (e.g., adding an image page to an existing document). As soon as a container becomes full, a new file system can be added.

Containers in a system are network-level resources. A name server holds container locations. Relocation of the volume set of a container to another jukebox, e.g., for load balancing, is possible through system management utility programs and can be achieved without changing any application's indexing database.

RetrievAll—The Megadoc Image Application Framework

Early Megadoc configurations required extensive system integration work. RetrievAll is the second-generation image application framework (IAF). The first generation was based on delivery of source of example applications. However, tracking source changes appeared to be too big of an issue and hampered the introduction of new base functionality.

In cooperation with European sales organizations, we formulated a list of requirements for a second-generation IAF. The framework must

1. Allow for standard applications. Standard applications, i.e., scan, index, store, and retrieve, cover a wide range of customer requirements in folder management. Tailoring standard applications can be accomplished in one day, without programming effort.

2. Be usable in system integration projects. The IAF must provide APIs for folder management, allowing the field to build applications with functionality beyond the standard applications by reusing parts of the standard applications.

3. Allow image enabling of existing applications. RetrievAll should allow the linkage of electronic image documents and folders with entities, such as order number or invoice number, in existing applications. Existing applications need not be changed and run on the image workstation using a terminal emulator running at the image workstation.

4. Accommodate internationalization. All text presented by the application to the end user should be in the native language of the user. RetrievAll should support more than one language simultaneously for multilingual countries.

5. Allow upgrading. A new functional release of RetrievAll should have no effect on the customer-specific part of the application.

6. Provide document routing. After scanning the documents, RetrievAll should route references to new image documents to the in-trays of users who need to take action on the new documents.

Image Documents in Their Production Cycle

Image documents start as hard-copy pages that arrive in a mailroom, where the pages are prepared for scanning. Paper clips and staples are removed, and the pages are sorted, for example, per department. An image batch contains the sorted stacks of pages. The scanning application identifies batches by a set of attributes. The scanning process offers a wide variety of options, including scanning one page or multiple pages, accepting or rejecting the scanned image for image quality control, batch importing from a scanning subsystem, browsing through scanned pages, and controlling scanner settings.

The indexing process regroups image pages of an image batch into multipage image documents. Each document is identified with a set of configurable attributes and optionally stored in one or more folders. Folders also carry a configurable set of attributes. On the basis of the attribute values, the document contents are stored in the document's storage location (container).

Many users of RetrievAll applications use the retrieve functions of the application only to retrieve stored folders and documents. Folders and documents can be retrieved by specifying some of the attributes. RetrievAll allows the configuration of query forms that represent different views on the indexing database. The result of a query is a list of documents or folders. For documents, the operations are view, edit, delete, print, show folder, and put in folder. The Megadoc editor is used to view and to manipulate the pages of the document including adding new pages by scanning or importing. For folders, the operations are list documents, delete, and change attributes.

Document Routing Applications

A RetrievAll routing application is an extension to a folder management application. A route defines how a reference to a folder travels along in-trays of users or work groups.

Systems Management

The following systems management functions support the RetrievAll package:

- Container management
- Security, i.e., user and group permissions
- Logging and auditing
- Installation, customization, tailoring, and localization

Architecture and Overview

As illustrated in Figure 5, the RetrievAll image application framework consists of a number of modules. Each module is a separate program that performs a specific function, e.g., scanning or document indexing. Each module has an API to control its functionality, and some modules have an end-user interface. Modules can act as building bricks under a control module. For example, an image document capture application uses

1. Scan handling, to let an end user scan pages into a batch.
2. Scanner settings, to allow the user to set and select the settings for a scanner. The user can save specific settings for later reference.
3. Batch handling, to allow the end user to create, change, and delete batches.

These three modules can operate together under the control of the scan control module and in this way form a document capture application. The scan control module can, under control of a main module, perform the document capture function in a folder management application.

Modules communicate by means of dynamic data exchange (DDE) interfaces provided in the Microsoft Windows environment. Each module, except the main module, can act as a server, and all modules can act as clients in a DDE communication.

Main Module Any RetrievAll application has a main module that controls the activation of major functions of the application. These functions include scanning pages into batches, identifying pages from batches into multipage image documents and assigning documents to folders, and retrieving documents and folders. The main module presents a menu to select a major function. The main module activates the control modules of the major functions in an asynchronous way. For example, the main module can activate a second major function, e.g., retrieve, when the first major function, e.g., identification, is still active.

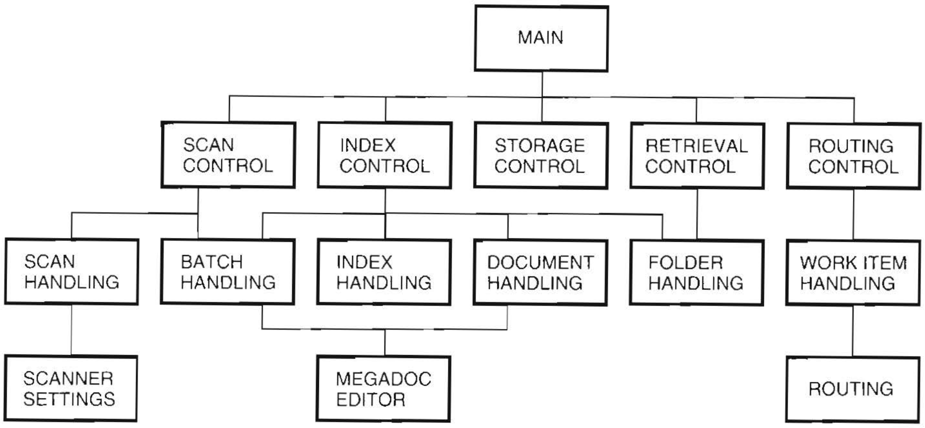


Figure 5 RetrievAll Module Overview

Control Modules Each major RetrievAll function has a control module that can run as a separate application. For example, when a PC acts as a scan workstation, it is not necessary to offer all the functionality by means of the main module. Control modules can be activated as a server through the DDE API with the main module as client or as a program item from a Microsoft Windows program group.

Server Modules All modules, with the exception of the main module, act as DDE server modules.

Configuration files hold environment data for each module. An application configuration file describes which modules are in the configuration. The layout of the configuration files is the same as the WIN.INI file used by the Microsoft Windows software, allowing the reuse of standard access functions.

Making an Application

An application can be made by selecting certain modules. Figure 5 gives an overview of the modules used for the standard folder management application. The installation program, which is part of the standard applications, copies the appropriate modules to the target system and creates the configuration files.

Modules can also be used with applications other than the standard ones. Image enabling an existing (i.e., legacy) application (see Figure 6), such as an order entry application where the scanned images of the orders should be included, entails the following:

- The existing application is controlled by a terminal emulator program running in the Microsoft Windows environment. This terminal emulator program must have programming facilities with DDE functions.
- While entering a new order into the system, the image document representing the order is on the screen. The function to include the image can be mapped on a function key of the emulator. Pressing the function key results in a DDE request to the identification function of the RetrievAll components. This DDE request passes the identification of the document (as known in the order entry application) to the identification function.

Summary

This paper has provided an overview of the many components and disciplines needed to build an effective image document management system. We discussed the details of the WORM file system, the storage manager technology, and the image application framework. Other aspects such as WORM peripheral technology, software compression and decompression of images, and the integration of facsimile and optical character recognition technologies were not covered.

From experience, we know that different customers have different requirements for image document management systems. The same experience, however, taught us to discover certain patterns in customer applications; we captured these patterns in the application framework. The resulting

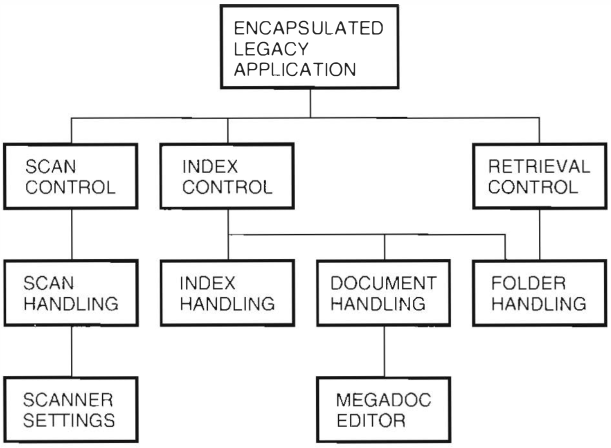


Figure 6 Image Enabling a Legacy Application

framework allows us to build highly customized applications with low system integration cost and short time to deployment. Future directions are in the area of enhanced folder management and integrated distributed work flows.

Reference

1. M. Bach, *The Design of the Unix Operating System,* ISBN 0-13-201757-1 (Englewood Cliffs, NJ: Prentice-Hall, 1986).

Mark F. Riley
James J. Feenan, Jr.
John L. Janosik, Jr.
T. K. Rengarajan

The Design of Multimedia Object Support in DEC Rdb

Storing multimedia objects in a relational database offers advantages over file system storage. Digital's relational database software product DEC Rdb supports the storing and indexing of multimedia objects—text, still frame images, compound documents, audio, video, and any binary large object. After evaluating the existing DEC Rdb version 3.1 for its ability to insert, fetch, and process multimedia data, software designers decided to modify many parts of Rdb and to use write-once optical disks configured in standalone drive or jukebox configurations. Enhancements were made to the buffer manager and page allocation algorithms, thus reducing wasted disk space. Performance and capacity field tests indicate that DEC Rdb can sustain a 200-kilobyte-per-second SQL fetch throughput and a 57.7-kilobyte-per-second SQL/Services fetch throughput, insert and fetch a 2-gigabyte object, and build a 50-gigabyte database.

To accommodate the increasing demand for computer storage and indexing of multimedia objects, Digital supports multimedia objects in its DEC Rdb relational database software product. This paper discusses the improvements over version 3.1 and presents details of the new features and algorithms that were developed for version 4.1 and are used in version 5.1. This advanced technology makes the DEC Rdb commercial database product a precursor of sophisticated database management systems.

Multimedia objects, such as large amounts of text, still frame images, compound documents, and digitized audio and video, are becoming standard data types in computer applications. Devices that scan paper, i.e., facsimile machines, are inexpensive and ubiquitous. Devices that capture and play back full-motion video and audio are just beginning to reach the mass market. Capturing these objects for use within a computer results in many large data files. For example, one minute of digitized and compressed standard TV-quality video requires approximately 50 megabytes (MB) of storage!

To date, relational databases have been used successfully in storing, indexing, and retrieving coded numbers and characters. Relational algebra is an effective tool for reorganizing queries to reduce the number of records, e.g., from 1 million to 70 records, that an application program must search to obtain the desired information. Other database features, such as transaction processing, locking, recovery, and concurrent and consistent access, are essential to the successful operation of numerous businesses. Electronic banking, credit card, airline reservation, and hospital information systems all rely on these features to query, maintain, and sustain business records.

However, although a business might have its numbers and characters organized, controlled, and managed in a computer database, maintaining the paper and film storage media associated with database records can be costly, both in dollars and in human resources. Some estimates place the worldwide data storage business at $40 billion, and as much as 95 percent of the information is stored on either paper or film. Currently, businesses such as insurance, banking, engineering, and medicine depend on human beings to manage the filing and retrieval of these extensive paper and film archives. Human error can result in the loss of paper and film. Clearly, scanning the paper, storing the information in a computer, and making this information available over computer networks is a better way to manage paper records. This scheme allows (1) multiple copies to be distributed at once; (2) a customer file to be electronically located and retrieved in seconds, whereas to materialize a paper folder can take days; and (3) properly programmed computers to maintain these types of information more efficiently and accurately than humans can.

The idea of eliminating paper-based storage of business records in favor of computer storage is long-standing. However, only recently have technical developments made it practical to consider capturing, storing, and indexing large quantities of multimedia objects. Storage robots based on magnetic tape or optical disk can be configured in the range of multiple terabytes (TB) at the low cost of 45 cents per MB. Central processors based on reduced instruction sets are getting fast enough to process multimedia objects without having to rely on digital signal coprocessors. Processor main memory can be configured in gigabytes (GB). Document management systems, which have thrived over the past few years, deliver computer scanning, indexing, storage, and retrieval across local area networks.

Until now, most multimedia objects have been stored in files. Document management systems generally use commercial relational database technology to store the documents' index and attribute information, where one attribute is the physical location of the file. This approach has several disadvantages: considerable custom software must be written and maintained to make the system appear logically as one database; application programs must be written against these proprietary software interfaces; a system based on both files and a relational database is difficult to manage; two backup-and-restore procedures must be learned and applied; and complications in the recovery process can occur, if the database and file system backups are executed independently.

Notwithstanding these disadvantages, storing multimedia objects in a relational database offers several advantages over file system storage.

- Coding an application against one standard interface structured query language (SQL) to store object attribute data as well as multimedia objects is easier than coding against both SQL to manage attribute data and a file system to store the multimedia object.

- The database requires only one tool to back up and monitor data storage rather than two to maintain the database and the file system.

- The database guarantees that concurrent users see a consistent view of stored information. In contrast to a file system, a database provides a locking mechanism to prevent writers and readers from interfering with one another in a general transaction scheme. However, a file system does offer locks to prevent readers and writers from simultaneous file access.

- The database guarantees, assuming that proper backup and maintenance procedures are followed, that no information is lost as a result of media or machine failure. All transactions committed by the database are guaranteed. A file system can be restored only up to the last backup, and any files created between the last backup and the system failure are lost.

In the sections that follow, we present (1) the results of an evaluation of DEC Rdb version 3.1 for its ability to insert, fetch, and process multimedia objects; (2) a discussion of the impact of optical storage technology on multimedia object storage; and (3) design considerations for optical disk support, transaction recovery, journaling, the physical database, language, and large object data storage and transfer. The paper concludes with the results of DEC Rdb performance tests.

Evaluation of DEC Rdb as a Multimedia Object Storage System

Given the premise that production systems need to store multimedia objects, as well as numbers and characters, in databases, the SQL Multimedia engineering team members evaluated the following DEC Rdb features to determine if the product could support the storage and retrieval of multimedia objects:

- Large object read and write performance

- Maximum large object size

- Maximum physical capacity available for storing large multimedia objects

The DEC Rdb product has always supported a large object data type called segmented strings, also known as binary large objects (BLOBs). The evolution from support for BLOBs to a multimedia database capability was logical and straightforward. In fact, the DEC Rdb version 1.0 developers envisioned the use of the segmented string data type for storing text and images in the database.

In evaluating DEC Rdb version 3.1, we came to a variety of conclusions about the existing support for storing and retrieving multimedia objects. Descriptions of the major findings follow.

The DEC Rdb SQL, which is compliant with the standards of the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO), and SQL/Services, which is client-server software that enables desktop computers to access DEC Rdb databases across the network, *did not* support the segmented string data type. Note that the most recent SQL92 standard does not support any standard large object mechanisms.[1] Object-oriented relational database extensions are expected to be part of the emerging SQL3 standard.[2]

The total physical capacity for storing large objects and for mapping tabular data to physical storage devices is insufficient. All segmented string objects have to be stored in only one storage area in the database. This specification severely restricts the maximum size of a multimedia database and thus impacts performance. One cannot store a large number of X-rays or one-hour videos on a 2- to 3-GB disk or storage area. Contention for the disk would come from any attempt to access multimedia objects, regardless of the table in which they are stored. Although multiple discrete disks can be bound into one OpenVMS volume set, thereby increasing the maximum capacity, data integrity would be uncertain. Losing any disk of the volume would result in the loss of the entire volume set.

The maximum size of the database that DEC Rdb can support is 65,535 storage areas, where each area can span $2^{32} - 1$ pages. That translates to 256 terapages (i.e., 256×10^{12} pages) or 128 petabytes (PB) (i.e., 128×10^{15} bytes). At a penny per megabyte, a 128-petabyte storage system would cost 1.28 billion dollars!

The largest BLOB that DEC Rdb can maintain is 275 TB (i.e., 275×10^{12} bytes). A data storage rate of 1 megabyte per second (MB/s) for motion video and audio translates into 8.7 years of video. However, as mentioned previously, the maximum size and the total number of objects that can be stored are limited. As part of system testing, we successfully stored and retrieved a 2-GB object in a DEC Rdb data field.

DEC Rdb uses a database key to reference individual segments stored in database pages. A BLOB belongs to only one column of one row of a relation. The database key value that locates the first segment is stored in the column of a table defined to represent the BLOB data type. DEC Rdb implements segmented strings as singly linked lists of segments. Therefore, version 3.1 must read a segment in order to find the next segment. This process has two disadvantages: (1) random positioning with a BLOB data stream is extremely slow, and (2) BLOB pages cannot be prefetched asynchronously. Figure 1 illustrates a DEC Rdb version 3.1 singly linked list segmented string implementation.

BLOB data transfer performance of DEC Rdb version 3.1 was promising. We were able to code a load test that sustained 65 kilobytes per second (kB/s); a fetch test sustained 125 kB/s. To put these measurements in perspective, DEC Rdb is capable of inserting more than one A4-size (210 millimeters [mm] by 297 mm, i.e., approximately 8.25 by 11.75 inches) scanned piece of paper per second and capable of fetching more than two A4-size pieces of paper per second. The test was conducted by writing and reading 50-kB memory data buffers to and from magnetic storage areas defined by the DEC Rdb software. This experiment ignores the overhead of network delays and compression.

DEC Rdb version 3.1 can write multiple copies of BLOBs, one to the target database storage area and one to each of the database journal files. The journal files provide for transaction recovery and

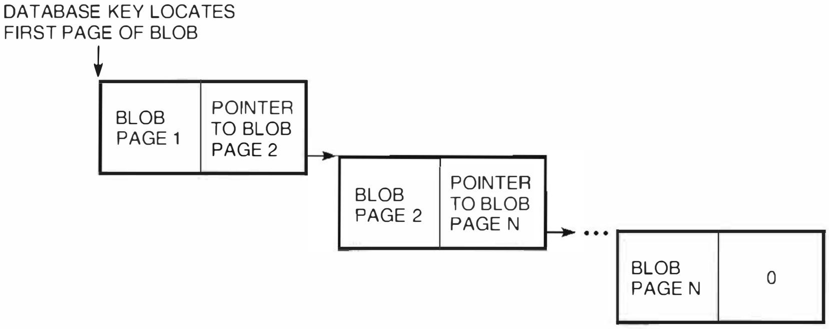


Figure 1 Rdb Version 3.1 Singly Linked List Segmented String Implementation

system failures, such as disk drive failures. Database journal files tend to be bottlenecks, because every data transaction is recorded in the journal. Therefore, writing large objects to journal files dramatically impacts both the size of the journal file and the I/O to the journal file.

The volume of storage required for most modest multimedia applications can be measured in terabytes. A magnetic disk storage system 1 TB in size is expensive to purchase and maintain. An alternative storage device that provided the capacity at a much lower cost was required. We investigated the possibility of using Digital's RV20 write-once optical disk drive and the RV64 optical library ("jukebox") system based on the RV20 drives. We quickly rejected this solution because the optical disk drives were interfaced to the Q-bus and UNIBUS hardware as tape devices. Since relational databases use tape devices for backup purposes only and not for direct storage of user data, these devices were not suitable. Note that physically realizing and maintaining a large data store is a problem for both file systems and relational databases.

DEC Rdb version 3.1 does not support large capacity write once, read many (WORM) devices, which are suitable for storing large multimedia objects. Version 3.1 has no optical jukebox support either.

Storage Technology Impact

When we evaluated DEC Rdb version 3.1, a 1-TB magnetic disk farm was orders of magnitude more expensive than optical storage. Large format 12- or 14-inch (i.e., 30.5- or 35.6-centimeter) WORM optical disks have a capacity of 6 to 10 GB. The WORM drives support removable media. These drives can be configured in a jukebox, where a robot transfers platters between storage slots and drives. A fully loaded optical jukebox, which includes optical disk drives and a full set of optical disk platters, of approximately 1-TB capacity costs about $400,000, i.e., $0.40 per MB. By comparison, Digital's RA81 magnetic disk drive, for example, has a capacity of 500 MB and costs $20,000. Thus, to store 1 TB of data would require 2,000 RA81 disk drives at a total cost of $40 million, i.e., $40.00 per MB!

How big is one terabyte? Assume, conservatively, that a standard business letter scanned and compressed results in an object that is 50 kB in size. Therefore, 1 TB can store 20 million business letters, i.e., 40,000 reams of paper at 500 sheets per ream. A ream is approximately 2 inches (51 mm) high, so 1 TB is equivalent to a stack of paper 80,000 inches or 6,667 feet or 1.25 miles (2 kilometers) high! The total volume of paper is 160 cubic yards (122 cubic meters). A 1-TB optical disk jukebox is about 3 to 4 cubic yards (2.3 to 3 cubic meters). Assuming TV-quality video, 1 TB can store 308 hours or approximately 12 days of video. Full-motion video archives suitable for use in the broadcast industry require petabytes of mass storage.

The gap between affordable and practical configurations of optical disk jukeboxes and magnetic disk farms has closed considerably since late 1992. Juxtaposing equal amounts (700 GB) of magnetic and optical storage, including storage device interconnects, installation, and interface software, reveals that magnetic disk storage is about five times more expensive than optical storage. The major disadvantage of optical jukebox storage is data retrieval latency related to platter exchanges. This latency, which is approximately 15 seconds, varies with the jukebox load and how data is mapped to different platters.

Mass storage technology, including device interconnects, combines different classes of storage devices into storage hierarchies. Storage management software continues to be a challenging aspect of large multimedia databases.

To provide 1 TB of mass storage capacity for relational database multimedia objects at reasonable cost, we conducted a review of third-party optical disk subsystems, hardware, and device drivers for VAX computers running the OpenVMS operating system. A characterization of the available optical disk subsystems revealed three basic technical alternatives.

1. Low-level device drivers provided by the drive and jukebox manufacturers.

2. Hardware and software that model the entire capacity of an optical disk jukebox as one large virtual address space.

3. Write-once optical disk drives interfaced as standard updatable magnetic disks. The overwrite capability is provided at either the driver or the file-system level, where overwritten blocks are revectored to new blocks on the disk. For example, consider a file of 100 blocks created as a single extent on a WORM device. When requested to rewrite blocks 50 and 51, the WORM file system writes the new blocks onto the end of all blocks written. The system also writes a new file header that contains three file extents: blocks 0 to 49

stored in the original extent; blocks 50 to 51 stored in the new extent; and blocks 52 to 100 stored as the third extent. Obviously, files that are updated frequently are not candidates for WORM storage. However, immutable objects, such as digitized X-rays, bank checks, and health-benefit authorization forms, are ideal candidates for WORM storage devices.

As a result of this investigation, we decided that using write-once optical devices, interfaced as standard disk devices, was the best solution to provide optical storage for multimedia object storage. This functionality is being met with commercially available optical disk file and device drivers.

In the future, WORM devices may be superseded by erasable optical or magnetic disks. However, experts expect that WORM devices, like microfilm, will continue to be useful for legal purposes.

Design Considerations

The tamperproof nature of WORM devices is an asset but causes special problems in database system design. The evaluation of DEC Rdb version 3.1 indicated that several features needed to be added to the DEC Rdb product to make it a viable multimedia repository. This section describes the design of the new multimedia features included in DEC Rdb versions 4.1 through 5.1.

Mass Storage

DEC Rdb version 4.1 supports WORM optical disks configured in standalone drive or jukebox configurations. DEC Rdb permits database columns that contain multimedia objects to be stored or mapped to either erasable (magnetic or optical disk) or write-once (optical disk) areas. The write-once characteristic can be set and reset to permit the migration of the data to erasable devices. No changes to application programs are required to use write-once optical disks, including jukeboxes.

The main design goals for WORM area support were to

- Reduce wasted optical disk space by taking into account the write-once nature of WORM devices
- Not introduce DEC Rdb application programming changes for WORM areas
- Maintain the atomicity, consistency, isolation, and durability (ACID) properties of transactions for WORM devices
- Maintain comparable performance, allowing for hardware differences between optical and magnetic devices

DEC Rdb uses the optical disk file system to create, extend, delete, and close database storage files on WORM devices. Although this approach uses the block revectoring logic in the optical disk file system, minimal space is wasted. When writing blocks to WORM devices, DEC Rdb explicitly knows that blocks can be written only once and bypasses the revectoring logic in the optical disk file system.

Nonetheless, DEC Rdb software could waste space in two major ways. First, when DEC Rdb creates a storage area on an erasable medium (e.g., a magnetic or erasable optical disk), the database pages are initialized to contain a standard page format, with page numbers, area IDs, checksums, etc. Preinitialized database pages help to determine corrupted database pages. However, preinitializing database pages on write-once media makes little sense. The second way in which DEC Rdb could waste write-once optical disk pages is to use storage allocation bit maps for space management (SPAM). SPAM pages are used to keep track of free and used pages. As records are added to and deleted from the database, the SPAM bit maps are constantly updated. SPAM pages are maintained within each database file. With write-once devices, a page can be used only once. Again, it makes no sense to update SPAM pages for write-once media.

To eliminate needlessly wasting space on write-once media, DEC Rdb does not preinitialize WORM pages. As a general rule, WORM areas should not contain any updatable data structures. DEC Rdb maintains WORM storage space allocation in the database root file. The database root file should always reside on a magnetic disk, because the root file is frequently updated and magnetic disks yield higher performance. The clusterwide object manager mechanism ensures that the pointer to the end of the written area is consistent across a cluster.

SPAM pages, although disabled for write-once areas, are in fact allocated anyway. The reason for allocating SPAM pages in a write-once area is to provide the ability to migrate the contents of the storage area to an erasable device. The SPAM pages simply need to be rebuilt to reflect the space utilization at the point of conversion.

This write-once characteristic was the basis for several enhancements to the buffer manager and page allocation algorithms. Given that a free WORM page has never been written to, the buffer manager simply materializes an initialized buffer in main memory for write operations without having to first read the page from disk. In the case of page allocation for magnetic disks, DEC Rdb must scan SPAM pages in search of enough free storage space to satisfy a write operation. The scanning algorithm is much simpler for write-once areas; to store new records, DEC Rdb allocates one more page at the end of the written portion of the area to a process. DEC Rdb maintains such allocated pages in a queue called the marked WORM page queue on a per-process basis. Whenever a WORM page is written to disk, that page is taken off the marked WORM page queue. An attempt to store a record checks the queue before allocating new WORM pages to storage. Facilities exist to allocate many WORM pages in one operation, thus minimizing the number of writes to the root file.

By explicitly taking into account the write-once characteristic of the device, DEC Rdb greatly reduces wasted space, keeping optical disk read and write performance high.

Transaction Recovery

To understand the discussion of transaction recovery, the concepts of first- and second-class records must be understood. Both alphanumeric records and BLOB segments are stored in database pages. Alphanumeric records are first-class records and thus have identities in tables; these records are the rows. First-class records are required to be on a medium that permits update (either magnetic disk or erasable optical disk). All relation tuples are first-class records. Second-class records, such as BLOBs, have no identities of their own. BLOBs can exist only within the domain of an alphanumeric record and are pointed to by first-class records. Second-class records may be located in WORM areas.

Multimedia objects can be stored as second-class records in either write-once or erasable areas. However, due to transaction recovery constraints, the rows of relations must be stored in magnetic disks as first-class records.

If an update transaction against the database is aborted, then the database must restore the state of all database areas to pretransaction state. Regardless of the transaction recovery scheme employed, e.g., hybrid undo-redo, the effects of an uncommitted transaction to write-once media may have to be undone.

By definition, a write transaction on write-once media, once complete, can never be undone. In cases where a transaction fails and the transaction has written data to a write-once area, DEC Rdb employs a logical undo operation. This operation de-references the database key that points to the BLOB data written as part of the failed transaction. An example helps to illustrate how the logical undo operation works.

1. Consider row R of table T, which contains a column defined as data type BLOB.
2. The BLOB storage map indicates that the large objects are stored in a write-once area.
3. A process starts a transaction and updates the row storing a BLOB in the write-once area.
4. For some reason the transaction aborts.
5. Recovery nullifies the value of the database key that locates the first page of the BLOB.

The write-once pages can never be reused and will never again be allocated. Nothing points to or references data written as part of an aborted transaction.

This transaction recovery scheme introduces the interesting phenomenon of WORM holes. Consider the following scenario:

- A write-once area has the first 106 pages written and allocated.
- Process X starts a transaction that writes a BLOB segment to the write-once area.
- Page 107 is allocated for process X.
- Later in time, process Y starts a transaction to store a BLOB in the same write-once area.
- Process Y causes pages 108 to 120 to be allocated, data is written, the transaction commits, and process Y disconnects from the database.
- At this point, process X decides to roll back its transaction.
- Page 107 remains in a preinitialized state.

Page 107 can never be allocated to store BLOB data. Recall that DEC Rdb manages space on write-once devices by maintaining an end-of-area pointer to keep track of pages that have been written. Zero-filled pages that will never be allocated are called WORM holes. WORM holes are interesting because DEC Rdb utilities, such as verify, expect to find all allocated pages in a standard format. The utilities have been modified to ignore empty pages on write-once areas.

Journaling Design Considerations

An effective database management system guarantees the recovery of a database to a consistent state in the event of a major system failure, such as media failure. Hence, full and incremental backups must be performed at regular intervals, and the database must record or keep a journal file of transactions that occur between backups. In DEC Rdb, the after image journal (AIJ) file records all transactions against the database since the last backup. Also, to recover from a system failure, the database must keep track of all outstanding or pending transactions. The recovery unit journal (RUJ) file records the state and data associated with all pending transactions.

Journal files are heavily utilized in a database management system. Contention for the journal files comes from every process that is updating the database. To be completely recoverable, the database management system must record BLOB data, as well as alphanumeric data, to both the AIJ and the RUJ files. Because multimedia objects are large, eliminating the need to write these objects to the journal files is desirable. The double-write transaction negatively impacts the performance of the application storing the object and taxes the journal file, one of the most burdened resources in the database.

As discussed in the Transaction Recovery section, DEC Rdb uses logical undo operations to undo aborted transactions. In addition to the minimal processing required to de-reference a database key pointing to the WORM area pages, DEC Rdb automatically disables RUJ log writes for WORM area records. This is another advantage of using WORM devices for multimedia objects.

Recording multimedia objects in the AIJ file is not so straightforward. DEC Rdb uses the AIJ file for media recovery, as well as for transaction recovery. By definition, keeping a media recovery journal forces twice the number of I/O operations, each to a separate device. DEC Rdb must write the multimedia object to the storage area designated for the multimedia object and write a copy of the object to the AIJ file. If the primary storage device that contains the object fails, the database administrator can apply the last full backup of the storage area, followed by any subsequent incremental backups, and roll forward through the AIJ journal file to recover the data. If a multimedia database is to be completely recoverable and consistent, then multimedia objects must be recorded in the AIJ file. Since they can never be erased, WORM optical disks might be the best devices to write an object (or a journal file) to. Even though a jukebox can misfeed and permanently damage the media, disks in a jukebox can be disk shadowed. The trade-off is doubling the I/O versus risking data integrity. Rather than legislate a policy, DEC Rdb permits applications to disable AIJ logging for BLOBs, thus transferring the risk to individual applications.

Database Physical Design Considerations

The original design of segmented strings specified a singly linked list, where the segments were written one at a time, as shown in Figure 1. When writing a new segment, the previous segment had to be updated with a pointer value that identified the location of the new segment. For example, to store a BLOB with two segments R1 and R2, the old algorithm stored R1, stored R2, and then modified R1 to point to R2. Although this algorithm does not waste space on a magnetic disk, it does waste space on write-once optical disk. Segment R1 must be rewritten to disk with a pointer to segment R2.

If we impose the dependency between the two stores that R2 must be stored before R1, the store dependency for BLOBs becomes a reverse order of segments. Storing segments in reverse order requires buffering all segments of a multimedia object. Whereas buffering the entire object in main memory may be feasible for small multimedia objects, main memory is not large enough to buffer audio and video data objects. The singly linked list method that DEC Rdb used prior to version 4.1 is not well suited for WORM devices. Therefore, we redesigned the format of BLOBs in WORM areas to eliminate the need to buffer large amounts of data.

The new design replaces the singly linked list with BLOB segment pointer arrays and BLOB data segments. The segment pointer array maintains a list of database keys that locate each segment, in order, for a BLOB, as illustrated in Figure 2. Because segment pointer arrays are stored as a singly linked list, the pointer arrays can become large. Application data is stored in BLOB data segments. The new method buffers and writes the BLOB segment pointers to disk after assigning the segmented string to a record.

Besides eliminating the waste problem for write-once devices, the segment pointer array has other advantages. DEC Rdb reads the pointer array into

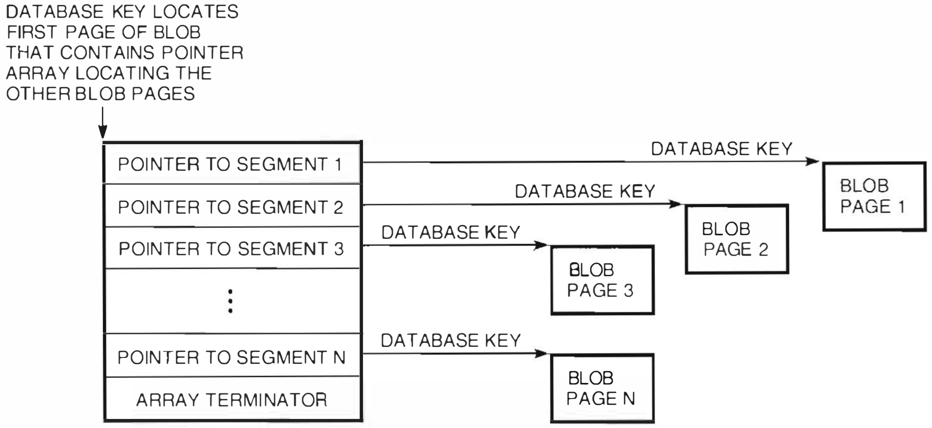


Figure 2 Rdb Version 4.2 Pointer Array Segmented String Implementation

memory when an application accesses a BLOB. DEC Rdb can, therefore, quickly and randomly address any segment in the BLOB. Also, DEC Rdb can begin to load segments into main memory before the application requests them. This feature benefits applications that sequentially access an object, such as playing a video game.

Storage Map Enhancements for BLOBs

Designers addressed several issues related to storage mapping. The major problems solved involved capacity and system management, jukebox performance, and the failover of full volumes.

Capacity and System Management DEC Rdb can map user data, represented logically as tables, rows, and columns, into multiple files or storage areas. Besides increasing the amount of data that can be stored in the database, spreading data across multiple devices reduces contention for disks and improves performance. However, as mentioned in the section Evaluation of DEC Rdb as a Multimedia Data Storage System, prior to DEC Rdb version 4.1, only one storage area could be used for storing BLOB data. All BLOB columns in the database were implicitly mapped into the single area, which severely limited the maximum amount of multimedia data that could be stored in DEC Rdb.

Prior to new multimedia support for BLOBs, DEC Rdb restricted the direct storage of a particular table column to one DEC Rdb storage area (i.e., file). This partitioning control is accomplished by means of the DEC Rdb storage map mechanism, as shown in the following code example:

```
Create storage map BLOB_MAP
    Store lists
        in RESUME_AREA
            for (PLACEMENT_HISTORY,
                 CANDIDATES.RESUME)
        in PHOTO_AREA
            for (CANDIDATES.PICTURE)
        in RDB$SYSTEM;
```

This code directs the BLOB data from the table PLACEMENT_HISTORY and the column RESUME of the table CANDIDATES to be stored in the area RESUME_AREA and the BLOB column PICTURE of the table CANDIDATES to be stored in the area PHOTO_AREA. The remaining BLOB data in the database is stored in the default RDB$SYSTEM area.

Restricting the storage of all BLOBs across the entire database schema to a single file or database area was clearly undesirable. The size of the area would be limited to the largest file that could be created by the OpenVMS operating system and the mass storage devices available. The limited mapping of one BLOB area mapped to one disk can be circumvented by using the OpenVMS system's Bound Volume Set mechanism. This mechanism allows *n* discrete disks to be bound into one logical disk. DEC Rdb can then create a single storage area on the logical disk that spans the bound set of disks.

However, although the volume set mechanism solves the problem of limited area mapping, serious limitations exist in the database system administration and recovery processes. All database-related facilities operate at the granularity of a database storage area. Thus, if one disk in a 10-disk volume set is defective, DEC Rdb would have to restore all

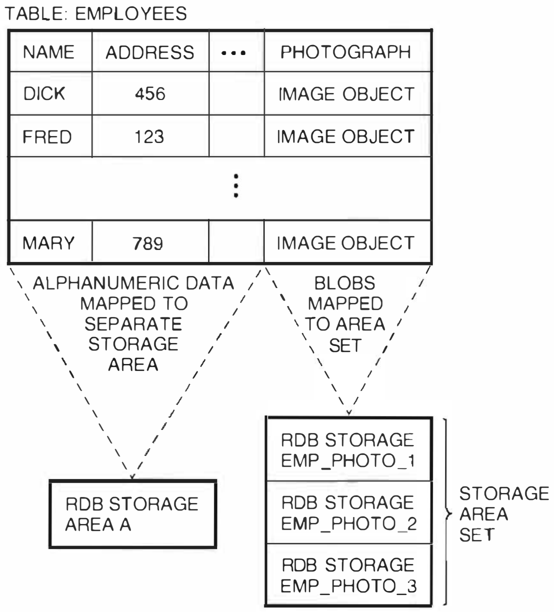
10 disks. Not only does restoring data on functioning disks waste processing time, but during the restore operation, applications are stalled for access at the area level. This situation introduces concurrency problems for on-line system operations.

DEC Rdb version 4.1 and successive versions solve the capacity problem by (1) permitting the definition of multiple BLOB storage areas, (2) binding discrete storage areas into storage area sets, and (3) providing the ability to map or to vertically partition individual BLOB columns to areas or area sets. Applications can set aside a disk or a set of disks for storing employee photographs, X-rays, video, etc. The alphanumeric data and indexes can be stored in separate areas as well. Figure 3 depicts the employee photograph column being mapped to the EMP_PHOTO_1, EMP_PHOTO_2, and EMP_PHOTO_3 storage area set. All alphanumeric data in the table EMPLOYEES is assumed to be mapped to storage area A.

Coding this example results in

```
Create storage map BLOB_MAP
   Store lists
      in (EMP_PHOTO_1,EMP_PHOTO_2,
          EMP_PHOTO_3)
         for (EMPLOYEES.PHOTOGRAPH)
      in RDB$SYSTEM;
```

This code directs the BLOB data, i.e., the column PHOTOGRAPH from the table EMPLOYEES, to be stored in the three specified areas EMP_PHOTO_1, EMP_PHOTO_2, and EMP_PHOTO_3.



NAME	ADDRESS	...	PHOTOGRAPH
DICK	456		IMAGE OBJECT
FRED	123		IMAGE OBJECT
	⋮		
MARY	789		IMAGE OBJECT

Figure 3 DEC Rdb BLOB Storage Area Sets

The ability to define multiple BLOB storage areas and to bind discrete areas into a storage set eliminates the BLOB storage capacity limitation in DEC Rdb. Consider the storage problem of storing 1 MB of medical X-rays as part of a patient record. Prior to DEC Rdb version 4.1, the limited one-BLOB storage area could store approximately 2,000 X-rays on a 2-GB disk device. The features included in version 4.1 allow the creation of a DEC Rdb storage area set that spans multiple disk devices. Also, adding storage areas or disks to a storage area set can expand the capacity initially defined for the column.

Jukebox Performance Problems When a storage area set is defined using the SQL storage map statement, DEC Rdb implements a random algorithm to select a discrete area or disk from the set to store the next object. Since multiple processes access multimedia objects across the entire set, a random algorithm that evenly distributes data across the disks in the area set reduces contention for any one disk.

Using a random algorithm to select from a set of platters in a jukebox is extremely inefficient. A jukebox comprises one to five disk drives with 50 to 150 shelf slots where optical disk media is stored. A storage robot exchanges optical disk platters between drives and storage slots. As described earlier, a full platter exchange—spin down the platter currently in the drive, eject the platter, insert a new platter, spin up the new platter—takes approximately 15 seconds. Each optical disk surface, i.e., side of a platter, is modeled as a discrete disk to the OpenVMS operating system. Consider, for example, ten storage areas defined on optical disks in the jukebox and mapped into a storage area set. All patient X-rays from a single table in the database are to be stored in this area set. Each new X-ray inserted in the database causes DEC Rdb to randomly select a disk surface in the jukebox, which probably results in a platter exchange. Consequently, each X-ray insertion takes 15 seconds!

The solution to the jukebox performance problem was not to eliminate random storage area selection, which works successfully with fixed-spindle devices. Rather, the solution was to accommodate an alternate algorithm that sequentially filled the disks in an area set. Using DEC Rdb, applications can specify random or sequential loading of storage area sets as part of the storage map statement.

Contention for a single optical disk in a jukebox is a far more desirable situation, with respect to latency, than causing one platter exchange per object stored.

When multiple users simultaneously issue requests to read multimedia objects stored in a jukebox, long delays occur, whether the storage area is loaded sequentially or randomly. Using a transaction monitor to serialize access to the database helps eliminate jukebox thrashing and improve the aggregate performance of the database engine.

Failover of Full Volumes The introduction of storage area sets gave rise to another problem: What happens when one area in the set becomes full? Normally, within the DEC Rdb environment, disk errors that result from trying to exceed the allocated disk space are signaled to the application so that the transaction can be rolled back (discarded). When related to storage area sets, however, the error is just an indication that a portion of the disk space allocated to the column has been exhausted and that processing should continue. Also, since multimedia objects tend to be exceedingly large, great amounts of data may have already exhausted cache memory and been written back to the WORM media, even though the database transaction has not committed. Handling such an error by signaling to the application and expecting the application to roll back and retry the transaction would result in the waste of a large number of device blocks that have already been burned. Thus, DEC Rdb had to implement a new scheme.

DEC Rdb now implements full failover of an area within the area set. Thus, when an area becomes full, DEC Rdb traps the error, selects a new area in the set, and writes the remaining portion of the BLOB being written to the new area. This area failover works whether the storage allocation is random or sequential. In addition, the area that is now full is marked with the attribute of full, and the clusterwide object manager of DEC Rdb maintains this attribute consistently throughout the cluster. Consequently, writers to the database will consider the area unavailable for future BLOB store operations. Further, the DEC Rdb database management utilities can remove the attribute if additional space is made available to the database area (e.g., if DEC Rdb moves BLOBs from area A to another copy of area A that resides on a device with twice the capacity).

Language Design Considerations

SQL, the ISO/ANSI standard relational database structured query language, is well suited to expressing queries against alphanumeric data yet hardly begins to address the needs of multimedia objects. Putting aside the fact that sampled data (i.e., a scanned image) is more difficult to query than coded data (e.g., text coded in ASCII), SQL cannot provide data compression and rendition capabilities for multimedia objects. Multimedia object processing is better suited to a language like C or C++. Ideally, SQL would support the ability to define objects and to associate methods with those objects. SQL3 is a new version of the SQL standard that the standards organizations are just beginning to work on. SQL3 contains the mechanism to define abstract data types and to execute external procedures as part of SQL statements. However, SQL3 will not become a standard for four to five years.

As discussed previously, DEC Rdb SQL lacks support for the segmented string or BLOB data type that was available in the Rdb relational engine. A new DEC Rdb SQL data type, LIST OF BYTE VARYING, was designed based on the native Rdb segmented string data type. The data access mechanism for the LIST OF BYTE VARYING data type is a list cursor, which operates like a table cursor—open the cursor, fetch segments of a BLOB, and close the cursor. This new data type with associated access mechanism was also added to SQL/Services. SQL/Services software enables remote clients on a network, such as personal computers, to attach to remote DEC Rdb databases. The ability to scroll or to randomly position the list cursor allows positioning at a particular data segment within the multimedia object stream without having to physically read through the entire data stream.

Although applications can program directly to list cursors, this interface was cumbersome and did not offer any object typing or processing. The list cursor mechanism does not present the straightforward byte-stream interface that is common in most file systems. Applications want to store objects, such as images and compound documents, not BLOBs. Data compression was another important consideration. Multimedia objects should be compressed on the client side of the network; then, compressed bits are transferred through the network, servers, and disks. The objects should be decompressed when they are to be rendered for display. Finally, the enormous size of multimedia objects saturates main memory resources on personal computers, so application developers must use disk storage to buffer as well as persistently store multimedia objects.

The limitations of the LIST OF BYTE VARYING data type and the list cursor data access mechanism led to the development of multimedia object extensions. SQL Multimedia is an object library that operates against SQL and SQL/Services. SQL Multimedia allows application developers to classify or type multimedia data types (e.g., IMAGE, TEXT, and COMPOUND_DOCUMENT) and to specify the data format within a type or class. Because no widely agreed upon multimedia object encodings or formats exist, we decided not to limit the types of data encoding or formats that could be stored in the database. For example, the database can store an image in Digital Document Interchange Format (DDIF) or Tagged Image File Format (TIFF). The option of defining a canonical encoding and format for each object class was too restrictive.

In both the SQL and the SQL/Services versions, the SQL Multimedia insert and fetch calls operate within the bounds of a transaction. All multimedia objects enjoy the same rights and privileges as alphanumeric data types in the database, with respect to concurrent access, recovery, etc.

A process that attaches to a DEC Rdb database can specify that an authorization identifier or a default identifier be created and referenced by the "RDB$HANDLE" symbolic label. A transaction can be started explicitly or a default transaction begins. To operate within the bounds of the default transaction, the SQL Multimedia routines required access to the default authorization identifier RDB$HANDLE. A new SQL compile time switch, for the SQL module language and precompilers, causes this identifier to be defined in a global address space. The SQL Multimedia routines can thus access the value of the identifier. If a distributed transaction identifier is not passed to the SQL Multimedia routines, the SQL Multimedia operation is executed using the default transaction.

SQL Multimedia improves the cumbersome list cursor interface by supporting the following object sources and destinations:

- The entire object sourced from or deposited to main memory
- The object buffered through main memory
- A file

SQL Multimedia handles file I/O operations across many different software environments, including the MS-DOS, Windows, Macintosh, ULTRIX, and OpenVMS operating systems. SQL Multimedia preserves file attributes on insert operations. For example, the Macintosh file system's resource fork, which contains the name and version of the application to be launched when the object is accessed by a user, is preserved. If another Macintosh user fetches the object to a local file, then SQL Multimedia restores the file including the resource fork. Assuming the second user has the same application, the user can now access and manipulate the multimedia object, e.g., a compound document or a QuickTime video file. Rules and default file organizations exist for the case where a user inserted a file from an OpenVMS system and another user causes the object to be fetched to a different client file system, say on a PC. Application programmers can direct SQL Multimedia to override the default file attributes.

Although SQL Multimedia handles disparate file system I/O, at present, it does not convert multimedia object formats or encodings. Images captured and stored in DEC Rdb in DDIF are delivered to each client in DDIF.

SQL Multimedia makes it easy for application programmers to insert and fetch compound documents to and from the database. The buffered I/O data stream conforms to Digital's Compound Document Architecture (CDA) stream management interface. Fetching a compound document using the buffered I/O interface, SQL Multimedia returns the address of a procedure entry mask, a data buffer pointer, and the buffer length. These returned arguments can be passed to the CDA viewer in the DECwindows environment. The viewer then repeatedly calls the SQL Multimedia buffer-fill procedure until the object has been transferred to the viewer and displayed.

In addition, SQL Multimedia provides object-specific processing for image and text objects. Disk image objects formatted according to DDIF and main memory objects formatted according to Digital's image toolkit DECimage Application Services (DAS) can be processed on either fetch or insert operations. SQL Multimedia leverages the capabilities of DAS software to provide image processing, e.g., compression, decompression, scaling, and dithering. When an image is inserted or fetched, SQL Multimedia object processing arguments permit the specification of image process steps and parameters. The DAS toolkit supports Comité Consultatif Internationale de Télégraphique et Téléphonique (CCITT) compression (a ubiquitous compression standard for facsimile machines) for bitonal images and Joint Photographic Experts Group (JPEG) compression (an ISO/ANSI standard) for multispectural images.

To improve application performance, SQL Multimedia can generate multiple rendered versions of an image that are stored in a single database field. Therefore, a user can store the original image, retaining its fidelity, and also store a miniature version of the image for fast access or browsing purposes. For example, consider a personnel application where 90 percent of the fetches for employee photographs are to be displayed in a passport-size format on an employee information form. If the capture portion of the application stored the original employee photograph and directed SQL Multimedia to generate and store a passport-size rendered version in addition to the original, at fetch time, the I/O operations required to transmit the image to the employee form would be reduced. Storing multiple rendered versions would also eliminate using CPU time to scale the fetched image.

System Testing and Evaluation

After the multimedia engineering of the DEC Rdb product was complete, we conducted several testing activities to determine the performance and capacity boundaries. The performance work presented is not complete but is offered as an indication of the multimedia object access capabilities of the DEC Rdb software.

In the debit credit domain, the Transaction Processing Performance Council (TPC) tests provide a standard procedure to measure the performance of one database as compared to another. However, no standard multimedia database performance tests exist. The performance of a DEC Rdb multimedia database is influenced by many variables, including the processor, mass storage medium, database design, object sizes, and workload. The performance data presented in this paper should be used only as a guide.

Performance Testing

For performance testing we used a VAX 6360 processor (relatively slow by today's standards) configured with 128 MB of main memory, an HSC50 storage interconnect processor with 16 RA70 magnetic disks, 6 RA92 magnetic disks, and 2 ESE20 solid-state disks. The total mass storage available for building databases was 10 GB. We evaluated the SQL performance of DEC Rdb version 4.2 Field Test 1 (FT1) and SQL Multimedia version 1.0 Field Test 2 (FT2), and generated the SQL/Services remote client data fetch and insert performance data for DEC Rdb version 4.1 Field Test 4 and SQL Multimedia version 1.0 FT2.

This performance data should be used as a guideline, because the field-test software contained implementation errors that affected performance but were corrected in the released products. As presented in Table 1, using the released version of DEC Rdb, we are able to sustain a 300-kB/s throughput from a magnetic disk DEC Rdb storage area, across an Ethernet network, to a DECstation 5240 workstation. This test demonstrates fetching a software motion pictures (SMP) video clip out of the database for display on an ULTRIX-based workstation.[3] Although the video was sampled at 15 frames per second, we can play back the video clip at 20 frames per second! The performance measured for an SQL/Services fetch was 57.7 kB/s, as shown in Table 2. We expect to conduct similar performance tests on a DEC 7000 AXP processor.

The performance test inserted and fetched 50-kB records. Fifty kilobytes is a conservative estimate of a compressed A4-size piece of paper, probably the most prevalent object to be stored in multimedia databases. For both the distributed SQL/Services client and the local SQL interface, 50-kB main memory buffers were the sources and destinations for the inserts and fetches.

We built several 50-MB databases, varying database design parameters such as page and buffer sizes, to determine the fastest set of parameters for the large object performance test. Using the largest page and buffer sizes yielded the best performance. The database table was organized into three columns: two key columns and a BLOB column. The BLOB column was mapped to a storage area set consisting of multiple magnetic storage disks.

After we established the best database organization, we built many 3- to 10-GB databases by

- Varying the number of processes executing insert and fetch operations
- Varying the number of tables in the database
- Varying the number of inserts and fetches per transaction

Table 1 SQL Performance

SQL Insert Performance

Number of Processes Performing Insert Operations	Number of Tables	Number of Inserts per Transaction	AIJ	Throughput (kB/s)
1	1	1	No	83.0
1	1	10	No	103.4
1	1	1	Yes	48.0
1	1	10	Yes	55.9
3	3	32	No	295.3
6	6	32	No	533.7
10	10	32	No	601.5

SQL Fetch Performance

Number of Processes Performing Fetch Operations	Number of Tables	Number of Fetches per Transaction	AIJ	Throughput (kB/s)
1	1	10	No	194.0
1	1	1	No	184.0
1	1	1	Yes	181.0
1	1	10	Yes	192.5

Table 2 SQL/Services Performance

SQL/Services Insert Performance

Number of Processes Performing Insert Operations	Number of Tables	Number of Inserts per Transaction	AIJ	Throughput (kB/s)
1	1	1024	No	44.0
4	4	32	No	91.9

SQL/Services Fetch Performance

Number of Processes Performing Fetch Operations	Number of Tables	Number of Fetches per Transaction	AIJ	Throughput (kB/s)
1	1	1024	No	57.7
4	4	32	No	142.3

- Enabling and disabling AIJ journaling
- Inserting and fetching from an SQL/Services client or using SQL for local database access

When we conducted the performance tests, the computer was dedicated to our task; no other activity was taking place. A simple contention test, where multiple readers simultaneously fetch objects from a single table, and a more complicated update test, where multiple writers are simultaneously updating one table, have yet to be fabricated and run.

To put some of the performance results presented in Table 1 into perspective: the tested configuration can sustain approximately 600 kB of insert bandwidth, which translates into twelve 50-kB

A4-size pieces of paper per second. Even a single process scanning paper at 103.4 kB/s can keep up with some of the fastest paper scanners available.

Also, scanning both sides of a compressed bank check (scanned at 200 dots per square inch) results in an object size of about 20 kB. Therefore, the particular configuration we tested could store 30 checks per second with multiple processes, and 6 checks per second with a single process.

Capacity Testing

We conducted two capacity tests. The first stored and fetched a 2-GB object in a DEC Rdb field, and the second built a 50-GB database. A 2-GB known pattern was generated in virtual memory. DEC Rdb wrote this object, with no AIJ, to a field in an empty database. The BLOB column was mapped to three disks, totaling 2.5 GB of storage. To avoid having to sustain storage area or file extensions, the storage area set was defined to be 2.3 GB. DEC Rdb was able to successfully insert and fetch the 2-GB object.

To demonstrate the capacity that could be achieved with SQL Multimedia, DEC Rdb, and optical storage, we built a 50-GB database. The hardware configuration consisted of the following:

- A VAX 4000 Model 500, with 6 GB of magnetic disk and 128 MB of main memory
- A Kodak Automated Disk Library Model 6800, with 100 GB of storage (with a maximum capacity of 1.2 TB)
- DEC Rdb version 4.2 Field Test 0
- SQL Multimedia version 1.0 FT2
- Perceptics LaserStar optical disk software

Starting with a backup of a 2-GB manufacturing database that was used by Digital's Mass Storage Group, DEC Rdb added an SQL Multimedia column to a table that contained over 550,000 rows. DEC Rdb then mapped the column to five platters, modeled as ten 9.5-million-block (5.1-GB) magnetic disks to the OpenVMS operating system, using the sequential load algorithm. An update table cursor was devised that returned between 2,000 to 3,000 rows. Using SQL Multimedia, DEC Rdb inserted images representing the disk assembly process until the storage was full.

Conclusion

The multimedia features that have been added to Rdb are in direct support of the increasing demand for computer data storage and indexing of multimedia object types (i.e., text, still images, compound documents, audio, and video). Relational database systems must expand mass storage device support, database physical database design, language functionality, and performance to manage the variety of today's information. The development of this advanced technology in Digital's DEC Rdb product provides desktop computer-to-optical disk jukebox integration by means of a commercial database. As multimedia technology matures, databases must address the need to store and index information beyond numbers and characters.

The work accomplished to support multimedia objects in DEC Rdb is just "the tip of the iceberg." Current multimedia capabilities are able to successfully manage the majority of document and still frame applications. However, improvement in capacity and performance are required before the database can serve multiple channels of video and audio data. As the SQL standard evolves to incorporate a more object-oriented mechanism, much of the SQL Multimedia functionality will migrate to using standard interfaces to define, operate on, and query abstract data types.

Acknowledgments

A large number of people from various disciplines contributed to the success of this multimedia database project, including Becky Jacobs, Michael Sawyer, John Lacey, Cheri Jones, Bruce Mills, Steve Hagan, Ian Smith, Susan Hillson, Peter Spiro, J. M. Smith, Jim Gray, Dave Lomet, Rudy Downs, Ken Cross (Perceptics), Chris Eastland, Mase Merchant, Scott Matsumoto, Paul Carmen (Eastman Kodak), Jim Lewis (Eastman Kodak), and Marilyn Gulliksen.

References

1. *American National Standard for Information Systems—Database Language—SQL,* ANSI X3.135-1992 (New York, NY: American National Standards Institute, 1992) and

   Information Technology—Database Language—SQL, ISO/IEC 9075:1992 (Geneva: International Organization for Standardization, 1992).

2. J. Melton, ed., *Database Language SQL (SQL3),* ISO/ANSI Working Draft, ANSI X3H2-93-091 and ISO/IEC JTC1/SC21/WG3/DBL YOK-003 (February 1993).

3. B. Neidecker-Lutz and R. Ulichney, "Software Motion Pictures," *Digital Technical Journal,* vol. 5, no. 2 (Spring 1993, this issue): 19-27.

General References

SQL Extensions

K. Meyer-Wegener, V. Lum, and C. Wu, "Image Management in a Multimedia Database System," *Proceedings of the IFIP TC 2/WG 2.6 Working Conference on Visual Database Systems*, Tokyo, Japan (1989): 497-523.

M. Stonebreaker, "The Design of the POSTGRESS Storage System," *Proceedings of the 13th International Conference on Very Large Databases*, Brighton, U.K. (1987): 289-300.

M. Stonebreaker and L. Rowe, *The POSTGRESS Papers*, Memorandum No. UCB/ERL M86/85 (Berkeley, CA: University of California, 1986).

Object Storage Management

M. Stonebreaker, "Persistent Objects in a Multi-Level Store," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, CO (1991): 2-11.

WORM Devices

D. Maier, "Using Write-Once Memory for Database Storage," *Proceedings of the ACM SIGMOD/SIGACT Conference on Principles of Database Systems (PODS)* (1982).

S. Christodoulakis et al., "Optical Mass Storage Systems and Their Performance," *IEEE Database Engineering* (March 1988).

S. Christodoulakis and D. Ford, "Retrieval Performance Versus Disk Space Utilization on WORM Optical Disks," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Portland, OR (1989): 306-314.

Storage Management for Large Objects

A. Biliris, "The Performance of Three Database Storage Structures for Managing Large Objects," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, CA (1992): 276-285.

Lawrence G. Palmer
Ricky S. Palmer

DECspin: A Networked Desktop Videoconferencing Application

The Sound Picture Information Networks (SPIN) technology that is part of the DECspin version 1.0 product takes digitized audio and video from desktop computers and distributes this data over a network to form real-time conferences. SPIN uses standard local and wide area data networks, adjusting to the various latency and bandwidth differences, and does not require a dedicated bandwidth allocation. A high-level SPIN protocol was developed to synchronize audio and video data and thus alleviate network congestion. SPIN performance on Digital's hardware and software platforms results in sound and pictures suitable for carrying on personal communications over a data network. The Society of Technical Communication chose the DECspin version 1.0 application as a first-place recipient of the Distinguished Technical Communication Award in 1992.

In late 1990, we began to design a software product that would allow people to see and hear one another from their desktop computers. The resulting DECspin version 1.0 application takes digitized audio and video data from two to eight desktops and distributes this data over a network to form real-time conferences. The product name represents the four major communication elements that unite into one cohesive desktop application, namely, sound, picture, information, and networks. The overall technology is referred to as SPIN. This paper first presents an introduction to conferencing and gives a brief overview of the framework on which SPIN was developed. The paper then details SPIN's graphical user interface. Although the high-level protocol (which is the application layer of the International Organization for Standardization/Open Systems Interconnection [ISO/OSI] model) that SPIN uses to synchronize distributed audio and video is proprietary, a general discussion of how SPIN uses standard data networks for conferencing is presented. Performance data for DECspin version 1.0 running on a DECstation 5000 Model 200 workstation with DECvideo and DECaudio hardware follows the discussion of network considerations. Finally, the paper summarizes the future direction of desktop conferencing.

Introduction to Conferencing

When the SPIN project started, standalone teleconferencing products were available but not for desktop computers. Typically, the products offered cost as much as $150,000, required scheduled conference rooms and operators, and needed leased telephone lines. These systems did not operate as part of a corporate computer data network but instead required dedicated, switched 56-kilobit-per-second (kb/s), T1 (1.5-megabit-per-second [Mb/s]), and T3 (45-Mb/s) public telephone components in order to operate. Originally designed as two-way conference units, these teleconferencing products later included hardware to multiplex several equally equipped systems. In addition, the enhanced systems included custom logic to implement a hardware compressor/decompressor (codec) that reduced digital video data rates sufficiently to use leased telephone lines.

During the last several years, other conferencing systems have been demonstrated. The Pandora research project by Olivetti Research resulted in an excellent desk-to-desk conferencing system. Although the Pandora system was expensive per user and did not use existing network protocols, it did prove the viability of using a digital conferencing system from one's office and demonstrated the natural progression from room conferencing to office conferencing. This system served as a good example for our own emerging desktop model, DECspin version 1.0.

Throughout this same period, several compression standards suitable for video capture and playback have evolved and been implemented. The Joint Photographic Experts Group (JPEG) industry-standard algorithm results in intraframe compression of frames of high-quality video (on the order of 25 to 1).[1,2] This algorithm is well suited for either single-frame capture or motion-frame capture of video information. This form of compression is most appropriate for real-time video capture and playback where low (i.e., frame-by-frame) latency is required.

The Motion Picture Experts Group (MPEG) standard results in interframe compression of motion video.[3] This algorithm is well suited for motion-frame capture of video because only the differences between successive frames are stored. Interframe compression is appropriate for video capture and playback where real-time low latency is not required.

The H.261 standard results in interframe compression of motion video that is most responsive to the demands placed on capturing live video for dissemination over low-bandwidth public telephone networks.[4] This compression is suitable for video capture and playback with reasonable latency but is not quite real-time in nature. H.261 is the standard used most in the teleconferencing systems on the market today.

Finally, the last few years have also witnessed the emergence of dramatic new base computer and network technologies. Reduced instruction set computer (RISC)-based workstations supply the needed processing power and I/O bandwidth to process large and continuous amounts of data, and fiber distributed data interface (FDDI) technology results in 100-megabit-per-second local area networks for the desktop. Consequently, the SPIN development project got under way to provide a novel and innovative software application that could take advantage of the powerful new systems and networks.

Overview of Underlying Hardware and Software

We came up with the SPIN project in response to the question: How can we communicate easily with graphics, video, and audio on the desktop as well as over both local and wide geographical area networks? Video help documentation, textual help, and audio help are used on the desktop to communicate how the application works. Sound, picture, graphics, and network elements are all woven together to provide better communication among conference participants.

Early in 1991, we received our first prototype of the DECvideo TURBOchannel frame buffer, which included the necessary hardware to input and capture an analog video signal, to digitize the signal, and to display the pixel information on the screen. The frame buffer was special in that it displayed 8-bit pseudocolor, 8-bit gray-scale, and 24-bit true-color graphical data simultaneously. This feature allowed captured video data to be displayed without data dithering.

Dithering is the process of converting each pixel of video data to a form that matches a limited number of available colormap entries. Most workstation frame buffers are 8-bit pseudocolor. Hence, digitized, 24-bit true-color video data for display would need pixel-by-pixel conversion. Algorithms exist that could be used to accomplish this conversion. However, a better SPIN conference, in terms of frame rate and picture quality, was achieved by performing no software dithering, thus relying on the ability of the DECvideo hardware to display 24-bit true-color video or 8-bit gray-scale video.[5] In addition, the DECvideo hardware could scale down the incoming video image in real time so that fewer pixels (i.e., less data) represented the original image.

Concurrently, SPIN used a DECaudio TURBOchannel card that could sample an input analog audio signal from a microphone and deliver an 8-kilohertz digitized audio bit stream. The DECaudio hardware could also convert a digital audio stream for output to an analog speaker or external amplifier. A DECstation 5000 Model 200 with DECaudio and DECvideo components provided the core hardware capability used in SPIN development work.

In addition to these new hardware capabilities, the SPIN effort needed new underlying base software capabilities. The DECvideo hardware required the Xv video extension to the X Window System to allow for the display and capture of video data. (The Xv extension was jointly developed by base system graphics and MIT Project Athena teams.) The DECaudio component used the AudioFile audio server, developed by Digital's Cambridge Research Laboratory, to capture and play back digital audio data.

A prototype software base was created to make fundamental measurements of video and audio data manipulation within the workstation and over a network. Testing the prototype over a 100-Mb/s FDDI network and a 10-Mb/s Ethernet network demonstrated that a conferencing product running over existing network protocols was possible.

The SPIN Application

SPIN is a graphical multimedia communications tool that allows two to eight people to sit at their desktop computers and communicate both visually and audibly over a standard computer data network. The user interface employs a telephone-like "push" model that allows a user to place an audio-only, video-only, or audio-video call to another desktop computer user. Here, the term "push" means that SPIN conference participants control all aspects of the digitized data they send onto a network. Thus, users can feel confident about the security of their audio and video information. A caller initiates all calls to other users, and a call recipient must agree to accept an incoming SPIN call. Because all data is in the digital domain, this model makes it almost impossible to use SPIN to eavesdrop on another person. Placing a wiretap on a person's call would involve intercepting network packets, separating data from protocol layers, and then reassembling data into meaningful information. If the network data were encrypted, interception would be impossible. SPIN also provides other communication services, such as an audio-video answering machine, messaging, audio-video file creation, audio help, and audio-video documentation. Figure 1 shows a screen capture of a SPIN session in progress, using the DECspin version 1.0 application.

The product is easy to learn and to use. The graphical user interface is implemented on top of Motif software. Motif provides the framework for the SPIN international user interface. A model was chosen in which all actions taken by a user are implemented by push buttons that activate pop-up menus. The SPIN application does not use pull-down menus, because they require language-specific text strings to identify the purpose of an entry and thus require translation for different countries. Also, pull-down menus are intended for short-term interaction, and SPIN menus usually require more long-term interaction. All push-button icons are pictorial representations of the intended function. For example, the main window has a row of five push buttons, each of which activates a specific function of the application and is shown in Figure 1.

In the main window, the first button from the left contains a green circle with a vertical white bar, the international symbol for exit. This button appears in the same location in each of the pop-up windows. It is used to exit the window or, in the main window, to exit the application.

The second button from the left is labeled with the communication icon. This button is used to select the call list shown in Figure 2. The call list contains the various buttons and widgets used to place a call to another user, to create and play back SPIN files, and to display a list of received SPIN messages, if any exist. The list provides a way to play and manage audio-video answering machine messages. For example, to place a call to another user on the network requires just three steps.

1. Enter the computer network name of the machine and user into SPIN's phone database as "user@desktop." A string representing something more understandable to a novice is also allowed, e.g., "user@desktop1.dec.com" becomes "user@desktop1.dec.com Firstname Lastname at Digital Equipment Corporation."

2. Select whether the call is to be sound only, picture only, or both. The toggle push buttons under the large note icon control audio select; those under the large eye icon control video select. Once the call is established, these buttons can be set or unset by clicking a mouse or using a touch-screen monitor and are useful for muting the audio portion or freezing frames of the video portion.

3. To establish a two-way network connection, press the call push button under the connection icon (which is labeled with two arrows going in opposite directions) that appears next to the desired call recipient. If the person called is logged on, a ring dialog box appears on the call recipient's screen and a bell rings. If the call recipient is not available, a dialog box appears on the caller's screen asking whether the caller wishes to leave a message. The caller can then choose to leave a message or not.

Depending on the individual settings, users can see and hear one another in multiple windows on the screen. To connect all conference participants in a mesh, press the "join" push button, which has a triangular icon.

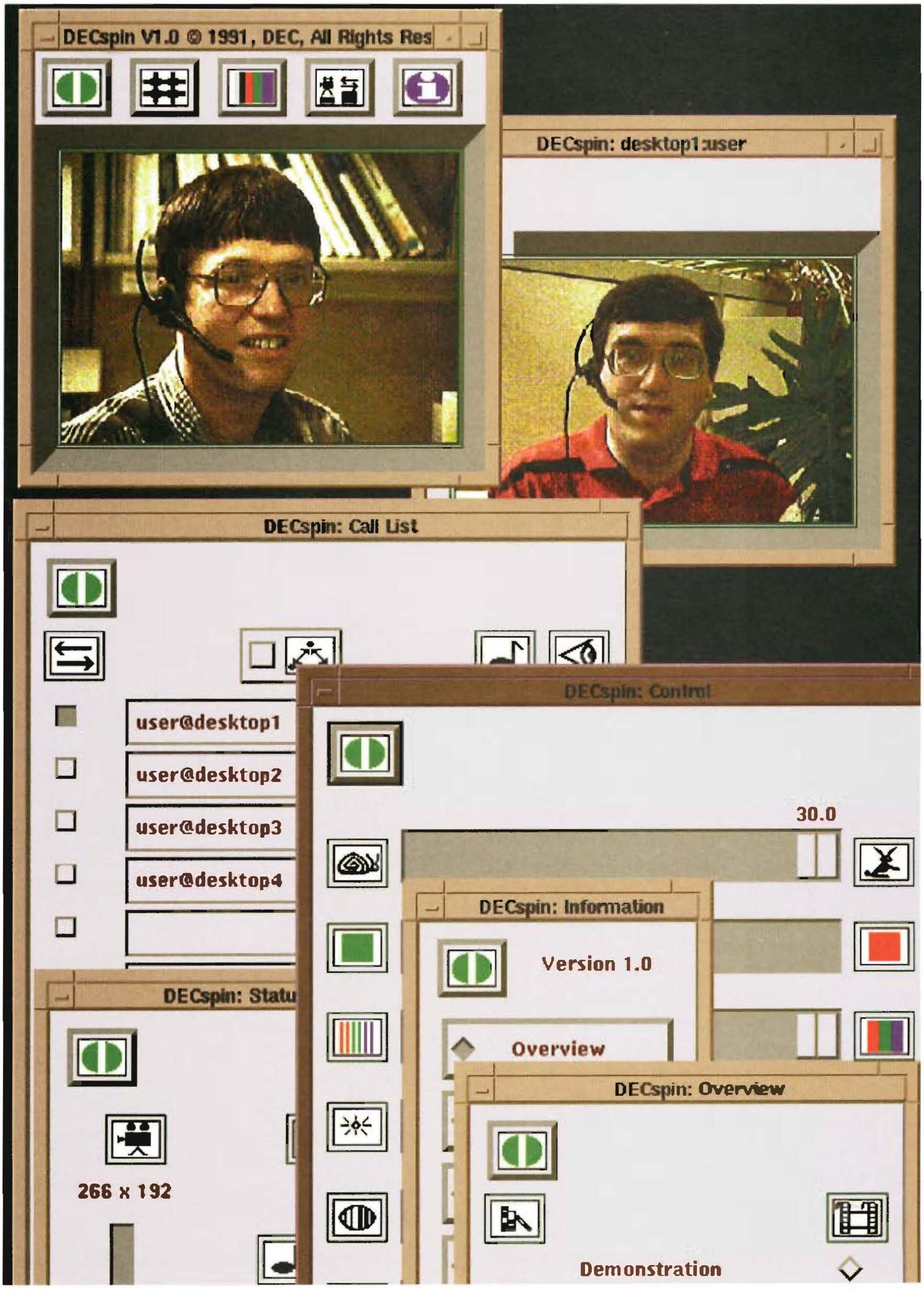


Figure 1 Sample SPIN Session

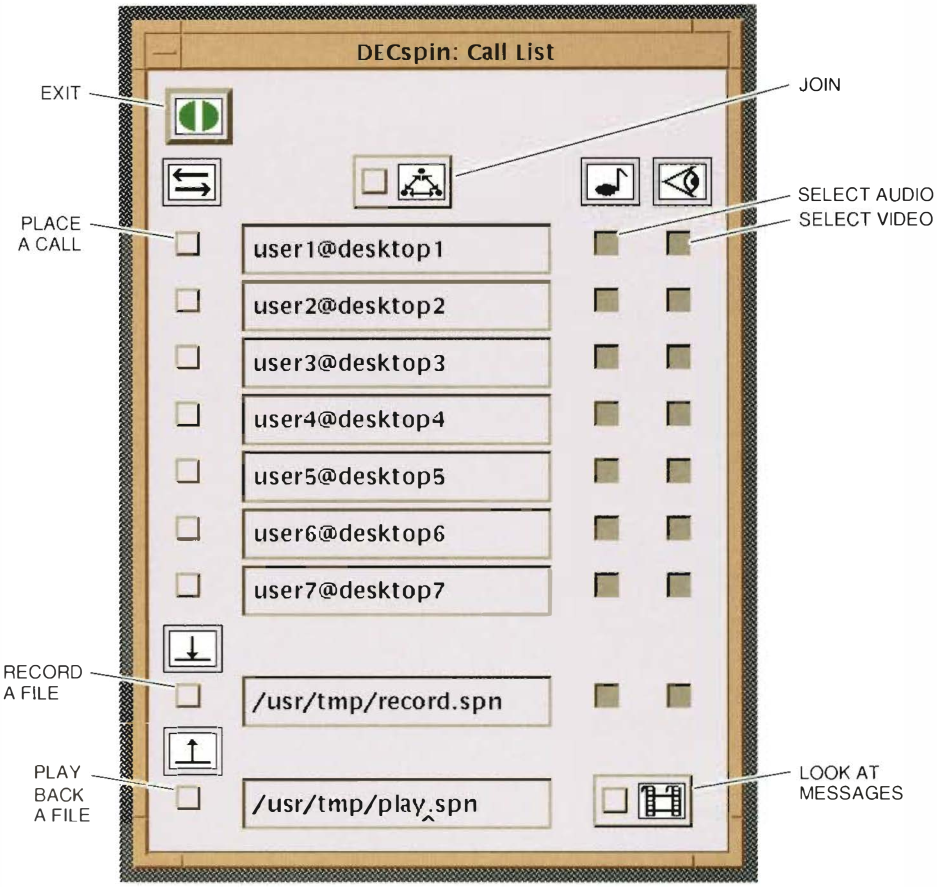


Figure 2 SPIN Call List Pop-up Window

Returning to the main window, the middle push button is the SPIN control button. As shown in Figure 3, the SPIN control pop-up window contains slide bars that, from top to bottom, allow the caller to set maximum capture frame rate, hue, color saturation, brightness, contrast, speaker output volume, and microphone pickup gain. At the bottom of the control window are buttons for selecting compression and rendering.

To the right of the control button in the main window is the status icon button. Pressing this button causes the status pop-up window shown in Figure 4 to appear. The status window displays, below the camera icon, the active size of the captured video area in pixels. Beneath these dimensions is a vertical slide bar that indicates the average frames-per-second (frames/s) capture rate sampled over a five-second interval. To the right of the camera icon is the connection icon, under which appears the number of active connections. Below this number are the sound and picture icons, under which appear the number of active audio connections and the number of active video connections, respectively. The second slide bar indicates the result of sampling the average outgoing bandwidth consumption (measured in Mb/s) of the application on the network. This measurement is also updated every five seconds.

Finally, the fifth push button (on the far right) in the main window is the information button. By pressing this button and selecting the type of online information desired, the user can access the documentation pop-up windows, as illustrated in Figure 5. Within each documentation window are several topics and two columns of toggle push buttons that can be used to obtain either textual documentation or video documentation. The video documentation comprises short videos that contain expert help about the operation of the application.

As a final level of help, all push buttons and widgets within the application have associated audio

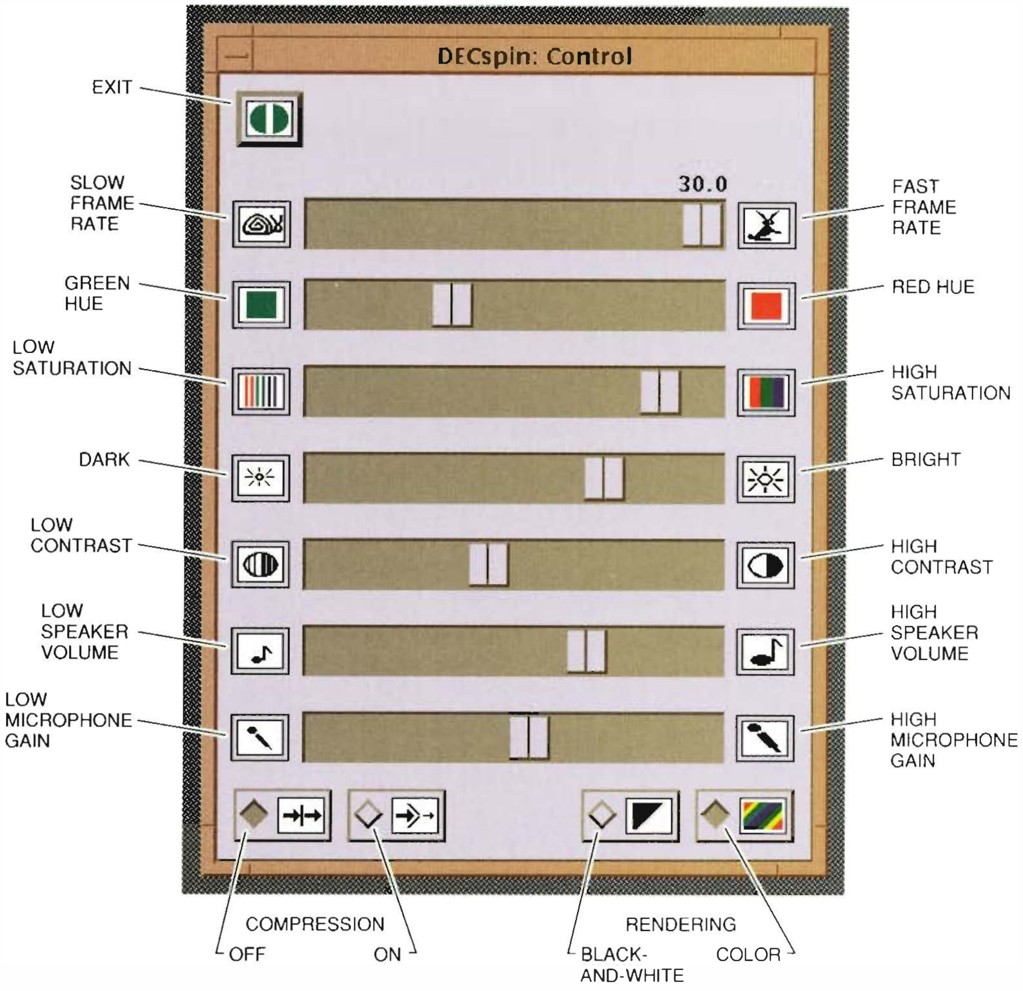


Figure 3 SPIN Control Pop-up Window

tracks that tell the user what the buttons and widgets do within their context in the application. To activate the audio tracks, the user must first select the button or widget and then press the Help key on the keyboard.

Network Considerations

SPIN uses standard data networks to transport the information that composes a conference. Data networks are usually private networks that a user community maintains. Such networks often include a number of individual networks joined together by bridges and routers. Unlike public telephone networks, which are most frequently used for phone calls, private networks are used for a variety of computer data needs, including file transfers, remote logins, and remote file systems. However, telephone networks often provide the long-distance lines used to make up private wide area data networks.

The use of data networks allows conferencing data to be treated as would any other type of data. SPIN requires no special low-level networking protocols to transmit its data and uses the transmission control protocol/internet protocol (TCP/IP) or the DECnet protocol. Also, SPIN requires no changes to existing operating systems. When performing the prototype work for the SPIN application, we were not certain whether the real-time nature of conferencing could be accomplished on inherently non-real-time networks and operating systems. Consequently, we developed a special high-layer synchronization conferencing protocol, called the SPIN protocol, that uses existing data networks.

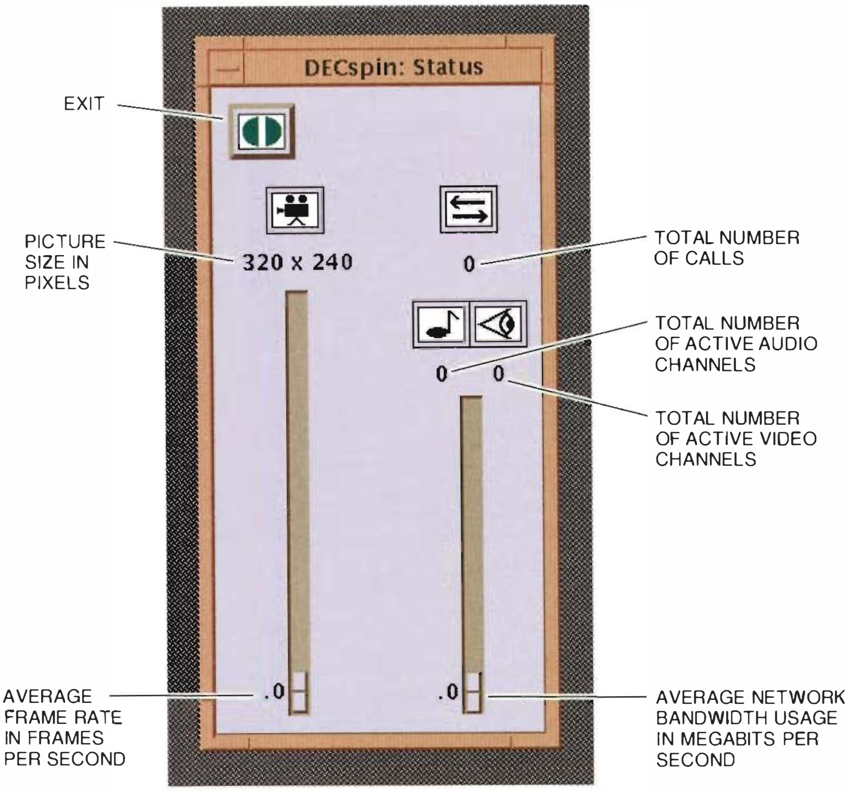

DECspin: Status
EXIT
PICTURE SIZE IN PIXELS
320 x 240
0
TOTAL NUMBER OF CALLS
TOTAL NUMBER OF ACTIVE AUDIO CHANNELS
0
0
TOTAL NUMBER OF ACTIVE VIDEO CHANNELS
AVERAGE FRAME RATE IN FRAMES PER SECOND
.0
.0
AVERAGE NETWORK BANDWIDTH USAGE IN MEGABITS PER SECOND

Figure 4 SPIN Status Pop-up Window

This protocol is responsible for the synchronization of audio and video information. The SPIN protocol monitors the flow of data to the network in order to alleviate network congestion when detected. As the network becomes congested, the protocol makes the decision to withhold further video data, since video is the largest consumer of network bandwidth. This withholding of video data is a key feature of the SPIN protocol, because it allows a conference to vary the video frame rate on a user-by-user basis. Thus, video bandwidth can scale to the lesser of either the bandwidth available or the number of frames/s of video bandwidth that a given platform can sustain.

If the withholding of video corrects the network congestion, video data is once again allowed in the conference. If not, the SPIN protocol delays audio data and stores it in a buffer until the network is able to handle this data. If the network outage lasts approximately 10 seconds, audio data is lost. Periods of audio silence are used as a means of recovery from periods of network congestion. Thus, variable video frame rates along with this treatment of audio data allow for the graceful degradation of a conference as the network becomes busy.

SPIN has been demonstrated over a variety of public and private data networks including Ethernet (10 Mb/s), FDDI (100 Mb/s), T1 (1.5 Mb/s), T3 (45 Mb/s), cable television (10 Mb/s, more correctly, Ethernet running over two 6-megahertz cable television channels), switched multimegabit data service (SMDS) (1.5 or 45 Mb/s), asynchronous transfer mode (ATM) (150 Mb/s), and frame relay (1.5 or 45 Mb/s). Some of these networks are local or metropolitan area technologies, i.e., local area networks (LANs), whereas others are wide area technologies, i.e., wide area networks (WANs), as illustrated in Figure 6.

Each type of network provides SPIN with different latency and bandwidth characteristics. SPIN makes corresponding adjustments to a conference to account for these differences and does not require a dedicated bandwidth allocation to carry

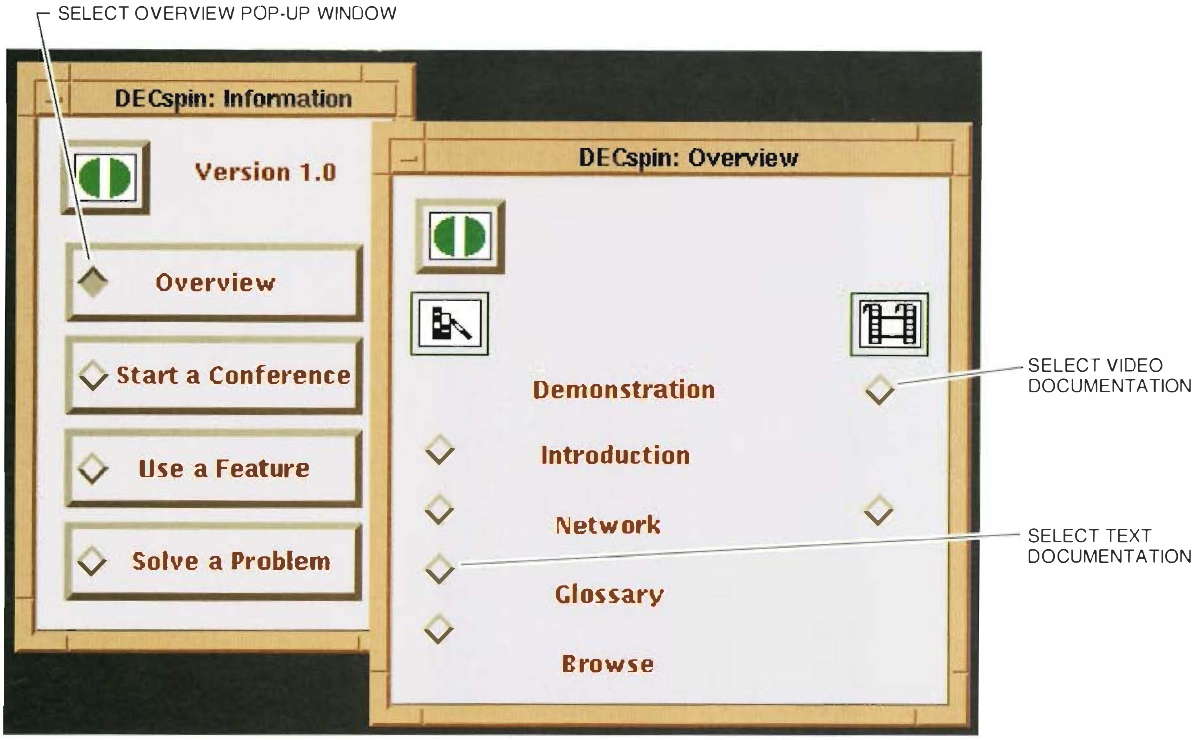


Figure 5 SPIN Information Pop-up Windows

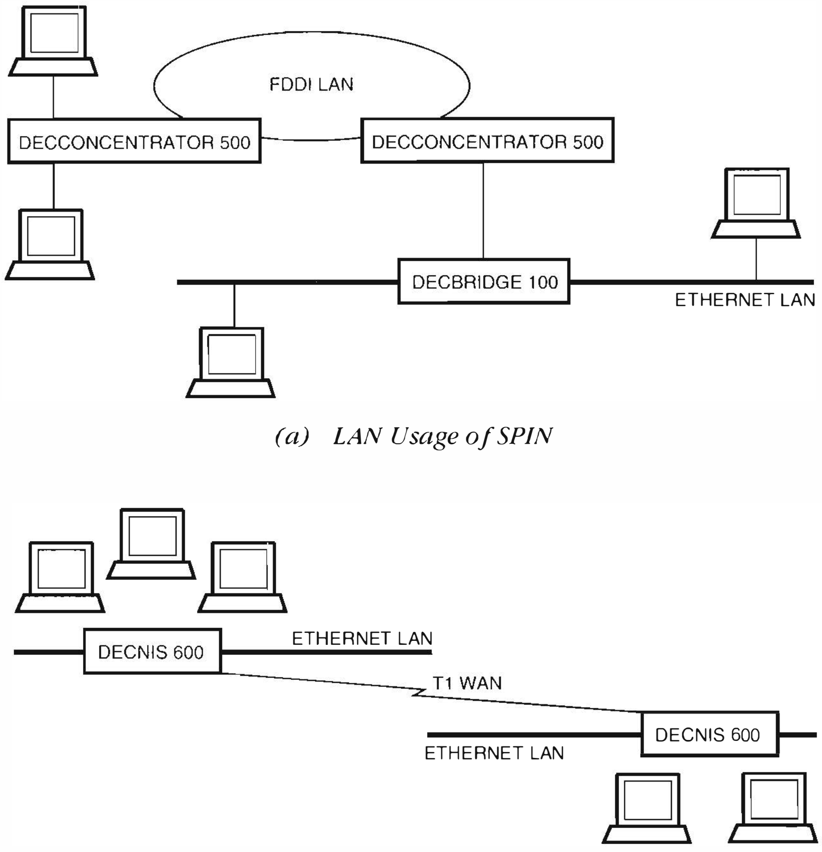
on a conference. If a given network supports bandwidth allocation, this feature only enhances SPIN's ability to deliver video and audio information.

WANs may use a router to interconnect two or more LANs. SPIN has been tested on a number of routers with mixed results, i.e., some routers correctly handle SPIN's bidirectional traffic pattern whereas others do not. Since some routers do not correctly handle bidirectional data traffic without packet loss, wide area routers must be individually tested with SPIN to verify proper operation. Some router problems were traced to the use of old firmware or software. Consequently, SPIN acted like a diagnostic tool in pointing out these problems. For example, running the SPIN application with audio only, across Digital's private IP network, yields varied results. Digital's IP network is an example of an open network, with routers from most router vendors. We traced most instances of poor SPIN performance to old or obsolete routers (some in service for the last six years without upgrades). These routers usually dropped packets when routing between adjacent Ethernet networks that were only 10 percent busy. After these routers were upgraded to the DECNIS family of routers, the SPIN application functioned correctly, even on congested networks.

To demonstrate daily use of SPIN, we created a metropolitan area network (MAN). Figure 7 shows the network topology, which spanned the states of New Hampshire and Massachusetts. The test bed allowed us to demonstrate our FDDI products, including end-station FDDI adapter cards, multimode FDDI wiring concentrators, and single-mode FDDI wiring concentrators. SPIN was used in 30 workstations, two of which were attached to large-screen projection units in conference rooms.

Performance

The conference quality achieved when running the SPIN application depends on many factors. The available network bandwidth, the processor speed, the desired frame-rate specification, the compression setting, the picture size, and how the pictures are rendered all affect the quality of the conference. Table 1 contains performance data for DECspin version 1.0 at various combinations of settings for these factors.



(a) LAN Usage of SPIN

(b) WAN Usage of SPIN

Figure 6 LAN and WAN Usage of SPIN

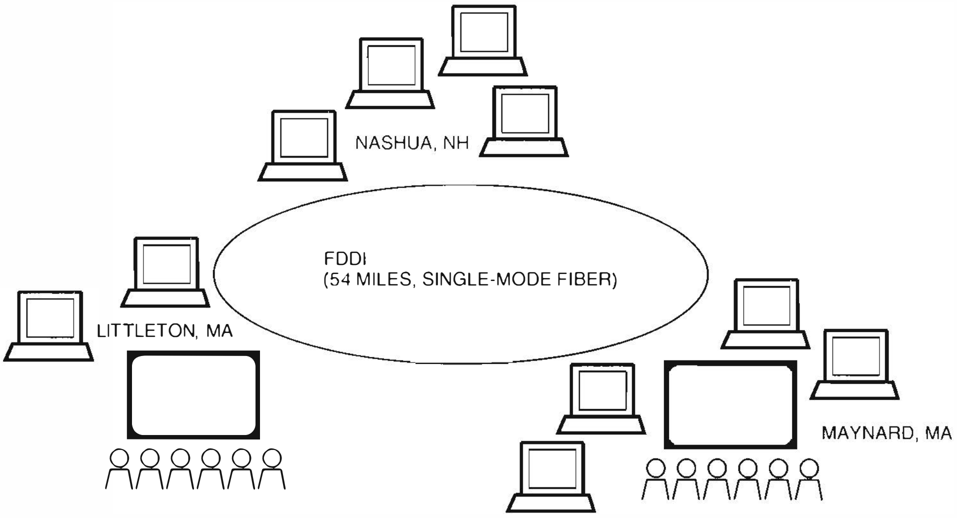


Figure 7 Digital's MAN Test Bed for SPIN

Table 1 SPIN Performance on a DECstation 5000 Model 200 with DECvideo and DECaudio Hardware

Width × Height (Pixels)	Render/Compression	Network	Frames/s (Bit rate in Mb/s)
256 × 192	Black and White/ No	FDDI	10 (4.0)
256 × 192	Black and White/ Yes	FDDI	16 (1.5)
256 × 192	Color/ No	FDDI	3 (5.0)
256 × 192	Color/ Yes	FDDI	10 (4.0)
160 × 120	Black and White/ No	FDDI	19 (3.0)
160 × 120	Black and White/ Yes	FDDI	25 (0.9)
160 × 120	Color/ No	FDDI	12 (6.0)
160 × 120	Color/ Yes	FDDI	19 (3.0)
256 × 192	Black and White/ No	Ethernet	9 (3.8)
256 × 192	Black and White/ Yes	Ethernet	16 (1.5)
256 × 192	Color/ No	Ethernet	2 (3.0)
256 × 192	Color/ Yes	Ethernet	8 (3.0)
160 × 120	Black and White/ No	Ethernet	19 (3.0)
160 × 120	Black and White/ Yes	Ethernet	25 (0.9)
160 × 120	Color/ No	Ethernet	8 (5.0)
160 × 120	Color/ Yes	Ethernet	19 (3.0)
Using a DECNIS Router (Ethernet-to-Router-to-T1-to-Router-to-Ethernet)			
256 × 192	Black and White/ No	T1	4 (1.4)
256 × 192	Black and White/ Yes	T1	15 (1.4)
256 × 192	Color/ No	T1	1 (1.4)
256 × 192	Color/ Yes	T1	4 (1.4)
160 × 120	Black and White/ No	T1	10 (1.4)
160 × 120	Black and White/ Yes	T1	25 (0.9)
160 × 120	Color/ No	T1	3 (1.4)
160 × 120	Color/ Yes	T1	10 (1.4)

As shown in Table 1, we tested SPIN performance using two basic picture sizes: 256 by 192 pixels and 160 by 120 pixels. The tests were performed over both Ethernet and FDDI networks for black-and-white and color cases. Also noted in the table is whether or not software compression was enabled for a specific test case. The far right column shows the frame rate achieved for the different combinations and also summarizes the network bandwidth consumed in each test. The table is presented primarily to give a sampling of the frame rate and, hence, the level of visual quality achieved for a specific combination of parameters. Frame rates affect an observer's ability to detect change within a sequence of frames. With a slow frame rate, the resulting video sequence may appear choppy and incomplete, whereas a normal frame rate (24 to 30 frames/s) leads to a smoothly varying video sequence with even continuity from one sequence to another. The frame rates in Table 1 below about 6 to 7 frames/s are considered low quality. Those in the 8-to-19-frames/s range are considered good quality, and those in the 20-to-30-frames/s range are high-quality video. The best cases in Table 1 are those that used software compression to deliver a pleasing frame rate with the least amount of network bandwidth consumed together with some degradation of individual frame quality. The software compression was tuned to provide nearly the same frame quality as the uncompressed case.

Table 1 also shows performance data measured using a DECNIS router. As noted earlier, wide area usage of SPIN depends on a router with correct algorithms for handling of bidirectional continuous stream traffic. The DECNIS family of routers can supply the full T1 bandwidth when presented with bidirectional SPIN traffic. Other routers on which SPIN was tested typically delivered only 25 to 50 percent of the T1 bandwidth. Note that this was only true on the particular routers we tested and that routers other than DECNIS routers may also be able to deliver full T1 bandwidth for this particular traffic pattern.

Hardware compression technology mentioned in the section Overview of Underlying Hardware and Software reduces the bandwidth requirements for conferencing. Experimentation with motion JPEG compression (using the Xv extension with compression functions on an Xvideo frame buffer board) has shown that at a resolution of 320 by 240 pixels, true-color frames can be used at 15 to 20 frames/s at a bit rate of just under 1.0 Mb/s. This bit rate produces a good- to high-quality conference with very low latency. H.261 and MPEG technology result in similar frame rates and picture size at about one-half the bandwidth but higher overall latency. Using motion JPEG as the example, high-quality conferences require about 1 Mb/s per connection. If all conferences are to be high quality, this bit rate allows 1 two-party conference on a T1 connection, 5 two-party conferences on an Ethernet segment, and 50 two-party conferences on an FDDI network. Using GIGASWITCH FDDI switches, more than 500 two-party conferences can take place simultaneously on a network. More users could be supported on T1, Ethernet, or GIGASWITCH networks, if lower-quality conferences are acceptable.

Conclusion

It became clear during the development and deployment of SPIN that high cost per user limits the widespread use of the application. The cost of video for DECspin version 1.0 adds about $8,000 to the price of a workstation. Audio for version 1.0 adds about $2,000 per workstation. These costs, which are prohibitive to most potential users of the technology, do not include the network cost impact.

Digital's Alpha AXP family of computers come with audio input and output hardware as part of the base workstation. In spring 1993, Digital released to the Internet community a version of DECspin that uses this hardware to carry on audio-only conferences and shows the user a voice waveform instead of a video image. This version eliminates the add-on hardware cost for audioconferencing. A new low-cost video option would go far to reduce the add-on cost for video and facilitate a wider use of the SPIN application.

The SPIN application and its associated protocol have been demonstrated on Digital and non-Digital computers, operating systems, and networks. In particular, SPIN has been shown on SPARC workstations running Solaris software. Additionally, SPIN has been demonstrated on a personal computer using the Microsoft Multimedia Extensions (MME) to Windows software. This platform provides a very large user community of potential SPIN users and dramatically drops the price per user compared with the original product. Interoperability among platforms and a common user interface give Digital a leadership position in this fast-forming market.

Today, high-quality conferencing can scale to hundreds of seats on a LAN with lower-quality conferencing scaling to larger, more geographically dispersed networks. Several factors will lead to the widespread use of this technology: better and less-expensive hardware, programmable codecs, and higher-speed and less-costly cross-country networks. Less-expensive video hardware allows many users to upgrade their systems to include video, while programmable compression technology allows users to achieve improvements in picture quality, compression transcoding, and lower network needs. Higher-capacity and less-costly cross-country networks allow more users to access conferencing services. Ultimately, even homes will have better computer connectivity and bandwidth. As these changes occur, and we believe they will, desktop conferencing can become the interactive telephone of the twenty-first century.

Acknowledgments

The authors wish to acknowledge and thank all the members of the close team that worked on the SPIN project and made the DECspin version 1.0 product a success. Key individuals in this effort were Diane LaPointe, Beverly Oliphant, Jonathan George, Garrett Van Siclen, and Jack Toto. We would also like to thank early supporters of the product efforts, including Jim Miller, Karl Pieper, and Jim Cocks. In addition, we extend our thanks to Walt Ronsicki, Videhi Mallela, Nathalie Rounds, and the rest of the team who established the FDDI test bed; to Dick Bergersen, who handled the quality assurance for DECspin version 1.0 and gave the team excellent feedback on the product; and to Tom Levergood and the other members of Digital's Cambridge Research Laboratory who gave us support and assistance in regard to the AudioFile audio server. Finally, we would also like to offer thanks to our management, particularly Bill Hawe and John Morse, who are strong advocates and supporters of our product efforts.

References

1. *Digital Compression and Coding of Continuous-Tone Still Images, Part 1, Requirements and*

Guidelines, ISO/IEC JTC1 Committee Draft 10918-1 (Geneva: International Organization for Standardization/International Electrochemical Commission, February 1991).

2. *Digital Compression and Coding of Continuous-Tone Still Images, Part 2, Compliance Testing,* ISO/IEC JTC1 Committee Draft 10918-2 (Geneva: International Organization for Standardization/International Electrochemical Commission, Summer 1991).

3. *Coding of Moving Pictures and Associated Audio,* Committee Draft Standard ISO 11172, ISO/MPEG 90/176 (Geneva: International Organization for Standardization, December 1990).

4. *Video Codec for Audiovisual Services at Px64 Kb/s,* CCITT Recommendation H.261, CDM XV-R 37-E (Geneva: International Telecommunications Union, Comité Consultatif Internationale de Télégraphique et Téléphonique [CCITT], August 1990).

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

General References

L. Palmer and R. Palmer, "Desktop Meeting," *LAN Magazine,* vol. 6, no. 11 (November 1991).

TURBOchannel Hardware Specification (Palo Alto, CA: Digital Equipment Corporation, TRI/ADD Program, 1990).

Open Software Foundation, Inc., *OSF/Motif, Programmer's Reference, Release 1.1* (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991).

R. Scheifler, J. Gettys, and R. Newman, *X Window System C Library and Protocol Reference* (Bedford, MA: Digital Press, 1988).

Peter C. Hayden

LAN Addressing for Digital Video Data

Multicast addressing was chosen over the broadcast address and unicast address mechanisms for the transmission of video data over the LAN. Dynamic allocation of multicast addresses enables such features as the continuous playback of full motion video over a network with multiple viewers. Design of this video data transmission system permits interested nodes on a LAN to dynamically allocate a single multicast address from a pool of multicast addresses. When the allocated address is no longer needed, it is returned to the pool. This mechanism permits nodes to use fewer multicast addresses than are required in a traditional scheme where a unique address is allocated for each possible function.

The transmission of digital video data over a local area data network (LAN) poses some particular challenges when multiple stations are viewing the material simultaneously. This paper describes the available addressing mechanisms in popular LANs and how they alleviate problems associated with multiple viewing. It also describes a general mechanism by which nodes on a LAN can dynamically allocate a single multicast address from a pool of multicast addresses, and subsequently use that address for transmitting a digital video program to a set of interested viewers.

Project Goals

The objective of this project was to design a mechanism suitable for providing the equivalent of broadcast television using computers and a local area data network in place of broadcast stations, airwaves, and televisions. The resulting system had to provide access to broadcast, closed circuit, and on-demand video programs throughout an enterprise using its computers and data network. The use of computer equipment installed for data transmission would eliminate the need to invest in cable TV wiring throughout a building.

The basic system would consist of two primary components. One computer, or set of computers, would act as a video server by transmitting video program material, in digital form, onto the data network. Other computers, acting as clients, would receive the transmitted video program and present it on the computer's display. Figure 1 depicts such a configuration.

The variety of video source material suggests that servers may be equipped in several ways. For example, accessory hardware can receive broadcast video programs; hardware and software can convert analog video into digital format; and hardware and software can compress the digital video for efficient use on a personal computer and data network.[1,2,3] Figure 2 shows a server equipped to handle different types of video program sources.

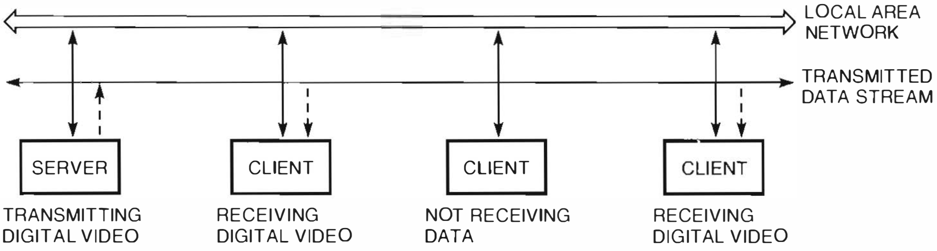


Figure 1 Client-server System for Video Data Transmission

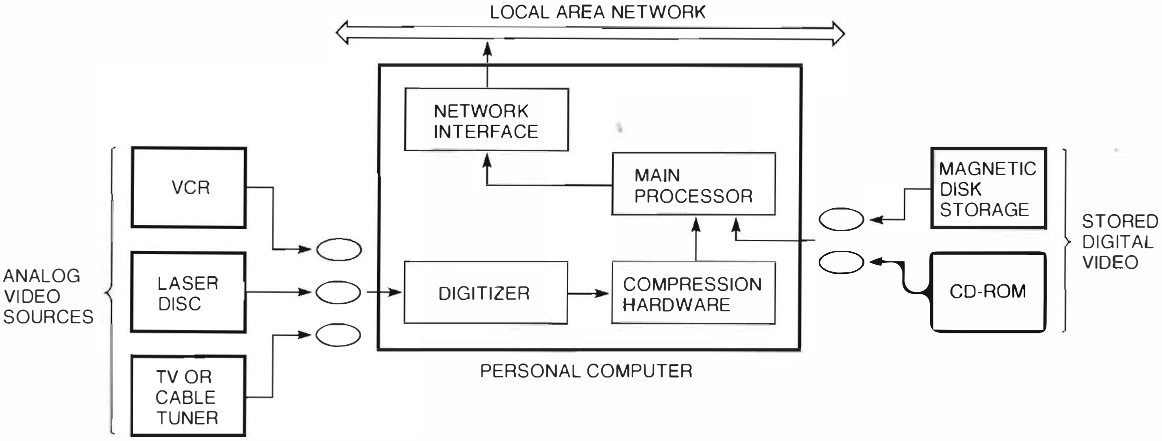


Figure 2 Types of Video Program Sources

Video program material is categorized as live, e.g., the current program broadcasting on a television network, or stored and played on demand, e.g., a recorded training session. In both cases, it is desirable for more than one client to be able to monitor or view the transmitted video program.

To implement the client-server system described above, many technical hurdles had to be overcome. This paper, however, focuses on one narrow but critical aspect, the addressing method used on the LAN for delivery of the digital video data. The characteristics of digital video and the need for multiple stations to receive programs from a wide range of possible sources combined to create some interesting challenges in devising a suitable addressing method.

Choosing an Addressing Method

To transmit digital video over a data network, an effective addressing mechanism had to be chosen that would satisfy the project's goals. Most LANs support three basic data addressing mechanisms: broadcast, unicast, and multicast.[4,5,6,7] Each method of transmitting digital video over a LAN has characteristics that are both attractive and undesirable.

Broadcast addressing uses a special reserved destination address. By convention, data sent to this address is received by all nodes on the LAN. Transmitting digital video data to the broadcast address serves the purpose of permitting multiple clients to receive the same transmitted video program while permitting the server to transmit the data once to a single address. Viewed another way, this convention is a significant disadvantage because all stations receive the data whether they are interested or not. Compressed digital video represents from 1 to 2 megabits per second of data; therefore nodes not expecting to receive the video data are impacted by its unsolicited arrival.[1,3] As a further complication, when two or more video programs are playing simultaneously, stations receive 1 to 2 megabits per second or more of data for each video program. This renders many systems inoperative. Furthermore, LAN bridges pass broadcast messages between LAN segments and cannot confine digital video data to a LAN segment.[8] As a result of these drawbacks, use of the broadcast address is unsuitable for transmission of digital video data.

Unicast addressing sends data to one unique node. The use of unicast addressing eliminates the problems encountered with broadcast addressing by confining receipt of the digital video data to a single node. This approach works quite well as long as only one node wishes to view the video program. If multiple clients wish to view the same program, then the server has to retransmit the data for each participating client. As the number of viewing clients increases, this approach quickly exhausts the server's capacity and congests the LAN. Because unicast addressing cannot practically support one server in conjunction with multiple clients, it too is unsuitable for transmission of digital video data.

Multicast addressing uses addresses designated to simultaneously address a group of nodes on a LAN. Nodes wishing to be part of the addressed group enable receipt of data addressed to the multicast address. This characteristic makes multicast addressing the ideal match for the simultaneous transmission of digital video data to multiple client nodes without sending it to uninterested nodes. Furthermore, many network adapters provide hardware-based filtering of multicast addresses, which permits high-performance rejection/selection of data based on the destination multicast address.[9] Because of these advantages, multicast addressing was selected as the mechanism for transmission of digital video data.

Multicast Addressing Considerations

Together with its advantages, multicast addressing brought significant problems to be overcome. The problems were in the assignment of multicast addresses to groups of nodes, all of which are interested in the same video program. If a single multicast address were assigned for all stations interested in receiving any video program, then only interested stations would receive data. All participating stations, however, would receive all programs playing at any given time. If multiple programs were playing, each station would receive data for all programs even though it is interested in the data for only one of the programs. The obvious solution is to allocate a unique multicast address for each possible program. The following sections examine various allocation methods.

Traditional Address Allocation

Traditionally, a standards committee allocates multicast addresses, each of which serves a specific purpose or function. For example, a specific multicast address is allocated for Ethernet end-station hello messages, and another is allocated for fiber distributed data interface (FDDI) status reporting frames.[10,11,12] Each address serves one explicit function. This static allocation breaks down when a large number of uses for multicast addresses fall into one category.

It clearly is not possible to allocate a unique multicast address for all possible video programs for several reasons. At any given time, hundreds of broadcast programs are playing throughout the world, and thousands of video programs and clips are stored in video libraries. Countless more are being created every minute. Assigning a unique address to each possible video program would exhaust the number of available addresses and be impossible to administer. Furthermore, it would waste multicast addresses since only those programs currently playing on a given LAN (or extended LAN) need an assigned address. A technique, therefore, is needed by which a block of multicast addresses is permanently allocated for the purpose of transmitting video programs on a computer network, and individual addresses are dynamically allocated from that block for the duration of a particular video program.

Dynamic Allocation Method

A dynamic allocation method should have several characteristics to transmit video programs on a LAN. These desired characteristics

1. Must be consistent with current allocation procedures used by standards bodies like the IEEE
2. Should be fully distributed and not require a central database (improves reliability)
3. Must support multiple clients and multiple servers
4. Must operate correctly in the face of LAN perturbations like segmentation, merging, server failure, and client failure

It is clearly desirable to use a dynamic allocation mechanism that does not require changes to the way addresses are allocated by standards committees. Changes to protocols only create another level of administrative complexity. Instead, a single set of addresses should be allocated on a permanent basis for use in the desired application. Drawn from a pool of addresses, these allocated addresses could be dynamically assigned to video programs as they are requested for playback. When playback was complete, the address would be returned to the pool.

Regardless of which allocation mechanism is adopted, it needs to support multiple servers and multiple clients. This implies that some form of cooperation exists between the servers to prevent multiple servers from allocating the same address for two different video programs. One node could act as a central clearinghouse for the allocation of addresses from the pool, but the overall operation of the system would then be susceptible to failure of that node. The preferred approach is a fully distributed mechanism that does not require a centralized database or clearinghouse.

LANs tend to be constantly changing their configurations, and nodes can enter and leave a network at any time. As a result, an allocation mechanism must be able to withstand common and uncommon perturbations in the LAN. It must accommodate events such as the segmentation of a LAN into two LANs when a bridge becomes inactive or disconnected, joining of two LANs into one when a bridge is installed or becomes reactivated, and failure or disconnection from the LAN at any time by both server and client nodes.

Other Multicast Allocation Methods

A variety of different group resource allocation mechanisms exist, and the one most nearly applicable to transmitting digital video over a LAN is used in the internet protocol (IP) suite. Deering discusses extensions to the internet protocols to support multicast delivery of internet data grams.[13] In his proposal, multicast address selection is algorithmically derived from the multicast IP address and yields a many-to-one mapping of multicast IP addresses to LAN multicast address. As a consequence, there is no assurance that any given multicast address will be allocated solely for the use of a single digital video transmission. This undermines the goal of using multicast addressing to direct the heavy flow of data to only those stations wishing to receive the data. Deering discusses the need for allocation of transient group address and alludes to the concepts presented in this paper.

Model for Dynamically Allocating Multicast Addresses

Given the overall goals of the project and the desired characteristics of the application, the following model was developed. It transmits digital video on a data network using dynamically allocated multicast addresses. First, simple operational cases on the LAN are described. Then complicated scenarios dealing with network misoperations are addressed.

It should be noted that the protocols described address the location of video program material as well as the allocation of multicast addresses for delivery of that material. Because of the one-to-one correspondence between video material and address allocation, it is convenient to combine these two functions into a single protocol; however, the focus of this paper remains on the address allocation aspects of the protocol.

Multicast Address Pool

This model assumes a set of *n* multicast addresses permanently allocated and devoted to it. The addresses are obtained through the normal process for allocation of multicast addresses through the IEEE. All clients and servers participating in this protocol use the same set of addresses. For the sake of this discussion, these addresses are denoted as A1, A2,...A*n*. Address A1 is always used by the participating stations for exchange of information necessary to control the allocation of the remaining addresses for use by the participating stations. The remaining addresses A2 through A*n* form the pool of available multicast addresses.

Server Announcements

All servers capable of transmitting digital video data continuously announce their presence and capabilities by transmitting a message at a predetermined interval; for example, a message is addressed to A1 every second. In these announcements, the servers include information identifying their general capabilities, data streams they are currently transmitting, and data streams they are capable of transmitting.

A server's general capabilities include its name and network address(es). Other useful information can also be announced, but it is not relevant to this discussion. To identify the data streams currently being transmitted, the server describes the data and the multicast address to which each data stream is being transmitted. In this way, it announces those multicast addresses that the station is currently using, along with a description of the associated video program. The data streams the server is capable of transmitting are identified by some form of a description of the data stream.

Identifying Servers and Available Programs

With each server continuously announcing the program material available for playback, clients wishing to receive a particular data stream can monitor the server announcements being sent to address A1. By receiving these announcements, a client can ascertain the address of each server active on the LAN, the data streams currently being transmitted by each server and the multicast address to which each is being transmitted, and the data streams available for transmission.

With a large repository of program material, it could easily become impractical to announce all available material. In this case, the announcements could be used only to locate available servers, and an inquiry protocol or database search mechanism could be used to locate available material more efficiently.

Once a client identifies a server that is offering the desired data stream, it can request that the server begin transmission. The client sends a message identifying the desired playback program material. In response, the server allocates a unique multicast address, includes the new material and multicast address in its announcement messages, and begins transmitting the program material.

Address Allocation and Tracking

Each server maintains a table containing the usage of each of the A2 to A*n* addresses. Each address is tagged as either currently used or available for use. When a server receives a client's request for transmission of a new data stream, the server selects a currently unused multicast address and includes the address and data stream description in its announcements of data streams currently being transmitted. After sending two announcements, the server begins transmitting the data to the chosen multicast address. Sending two announcements before beginning transmission provides client nodes with ample time to ascertain the address to which the data will be sent and to enable reception of the video program.

In addition to sending announcement messages, the servers also listen to the announcements from other servers to keep track of all multicast addresses currently in use on the LAN. Each time a server receives an announcement message from another server, it notes the addresses being used and marks them all as used in its table. This prevents a server from allocating an address already used by another server and eliminates the need for a central database or clearinghouse.

If a server observes that it is using the same address as another server, then the server moves its data transmission to another address if and only if its node address is numerically lower than the other server's node address. The new address is allocated exactly as it would be if the server were beginning to transmit the data stream for the first time. This algorithm resolves conflicts where two or more servers choose the same available multicast address at the same time. In addition, it resolves a similar conflict that occurs when two separate LAN segments become joined and two servers suddenly find they are using the same multicast address.

Clashing allocations of multicast addresses can be held to a minimum if servers allocate an address at random from the remaining pool of addresses rather than all servers allocating in the same fixed order.

Identifying and Stopping Playback

After a client requests playback of new material, it can then examine the server's announcements, and when the desired data stream appears as being transmitted by the server, the client can begin receiving data from the advertised multicast address. At this point, any other client stations on the LAN can also receive the same video program by enabling receipt of the same address.

When no more clients wish to view a particular program, a mechanism is needed to inform a server to stop transmission and return the associated address to the free pool. Two alternative approaches were considered to stop playback; one was chosen for several reasons.

In the first approach, each server tracks the number of clients that have requested a particular program by simply counting the number of requests for that program. In addition, clients are required to notify the server when they are finished viewing. The server then continues to transmit the material until all interested clients have indicated they are no longer interested in viewing. This approach has two problems. If a viewing client node is reset or disconnected, or if its message to end viewing is lost, the server could lose track of the number of viewing clients and never stop playing a particular program. The second problem, which is more of a nuisance, is that clients have to request playback of a program even if it is already playing to enable the servers to track the number of viewers.

In the preferred approach, interested clients periodically remind the server that they wish to continue viewing the program. Servers then simply keep playing the material until no client expresses interest for some period of time. For example, clients could reiterate their interest in a program every second, and a server could continue transmitting a requested program until it did not receive a reminder for 3 seconds. This time lapse would accommodate lost reminder messages from clients, and client failure would result in transmission termination within 3 seconds. In addition, when all clients had finished viewing the material, the server, multicast address, and consumed network bandwidth would be released within 3 seconds, making them available for other uses. Selection of the actual timer value depends on the desired balance between ongoing consumption of network resources (bandwidth and multicast addresses) after all receiving parties have stopped viewing the data, and network, end system, and server resource consumption caused by more frequent reminder messages.

Changing Multicast Addresses

Aside from receiving and processing the data for a video program, client stations must also continue to examine the server announcement messages and remain alert to possible changes in the multicast address to which the received program is being transmitted. As noted above, address allocation can change at any time due to merging of LAN segments or duplicate allocation by two servers. Anytime a client notes a change in address, it must stop receiving data on the previous address and resume receiving with the new address. A momentary disruption in playback is likely to occur, but such disturbances are infrequent because only merging LANs cause duplicate allocations of addresses in the middle of playback.

Under the circumstances described earlier, a client can find itself receiving two data streams on the same multicast address for some finite time period until the servers resolve the allocation of that address. Clients can gain immunity to this situation by noting the source address of the server that originally provided the data stream, and discarding all data received on the multicast address that is not from the source address. With this improvement, clients can easily distinguish the data stream of interest from another which might momentarily appear addressed to the same multicast address.

The allocation and resolution of multicast address use can be improved if servers send their announcements at an increased rate for some time period after a new data stream begins transmitting or when a data stream changes address. Such accelerated announcements permit client stations to more quickly identify the address of a requested data stream, and more quickly identify when a data stream has moved from one address to another. They also permit servers to more quickly identify instances of clashing multicast addresses and resolve them. For example, the announcement interval could be increased from 1 second to one-quarter second for a 2-second duration and resumed at 1-second intervals.

Extension to Interconnected LANs

The described protocols and allocation methods function correctly across multiple LANs interconnected by bridges since bridges nominally forward multicast traffic. Many bridge implementations permit management control over the forwarding of multicast data. This can unintentionally interfere with the desired operation of this protocol, but it can also as serve as a useful tool to confine data traffic to particular LAN segments. Another practical consideration in the particular application described here is the ability of a bridge to forward the large amounts of data traffic involved in digital video without detrimentally impacting the time-dependent nature of the data.

Extending the protocols to a wide area network is a more difficult procedure. Routers do not forward multicast traffic, but they could if used as proxy nodes between LANs. Router forwarding performance tends to be even lower than bridge forwarding rates, which discourages the operation of this system over a router.

Conclusions

Dynamic allocation of multicast addresses is critical to enable features such as the continuous play of full motion video over a network with multiple viewers. It is not feasible (or at least is very difficult) for a server to transmit a data stream individually to all clients wishing to receive it. If, on the other hand, the desired data stream is transmitted to the broadcast address, all stations on the LAN have to receive an enormous volume of data whether they are interested or not. It is highly desirable not to inundate uninterested clients with video data streams, but to send them to clients that want to receive specific video data streams in which they are interested.

Multicast addresses are well suited (in fact designed) for transmission to some arbitrary group of stations. To prevent a client that is receiving one video stream from being inundated by other video streams, a unique multicast address is required for each unique data stream. Since there are infinite individual data streams to choose from, it is impossible to allocate a unique multicast address for every data stream. A mechanism to allocate a unique multicast address from a finite set of addresses for the duration of the data stream is the ideal choice. The described mechanism also has the attractive characteristic that it is completely distributed; there is no central agent for allocation of multicast addresses; therefore it is more reliable as servers join and leave the LAN.

Although transmission of digital video data has prompted this system design, the basic mechanism for dynamically allocating multicast addresses can be applied to any application with similar needs.

Acknowledgments

I would like to acknowledge the assistance of John Forecast of Digital's Networks and Communications Group in enumerating the necessary pathological conditions in this work and for acting as a sounding board for proposed solutions.

References

1. K. Harney, M. Keith, G. Lavelle, L. Ryan, and D. Stark, "The i750 Video Processor: A Total Multimedia Solution," *Communications of the ACM*, vol. 34, no. 4 (April 1991).
2. G. Wallace, "The JPEG Still Picture Compression Standard," *Communications of the ACM*, vol. 34, no. 4 (April 1991).
3. D. Le Gall, "MPEG: A Video Compression Standard for Multimedia Applications," *Communications of the ACM*, vol. 34, no. 4 (April 1991).
4. *Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specification* (New York: The Institute of Electrical and Electronics Engineers, Inc., 1986).
5. *Fiber Distributed Data Interface—Token Ring Media Access Control* (New York: American National Standards Institute, 1987).
6. *Token Ring Access Method and Physical Layer Specifications* (New York: The Institute of Electrical and Electronics Engineers, Inc., 1986).
7. *Token-Passing Bus Access Method and Physical Layer Specifications* (New York: The Institute of Electrical and Electronics Engineers, Inc., 1986).
8. *Local Area Network MAC (Media Access Control) Bridges,* IEEE Standard 802.1(d) (New York: The Institute of Electrical and Electronics Engineers, Inc., 1990).
9. *MC68838 Media Access Controller User's Manual* (Phoenix, Arizona: Motorola, Inc., 1992).
10. *Logical Link Control,* ANSI/IEEE Standard 802.2-1985, ISO/DIS 8802/2 (New York: The Institute of Electrical and Electronics Engineers, Inc., 1985).
11. *A Primer to FDDI: Fiber Distributed Data Interface* (Maynard, MA: Digital Equipment Corporation, Order No. EC-H0750-42 LKG, 1991).
12. *FDDI Station Management—Draft Proposed American National Standard* (New York: American National Standards Institute, June 25, 1992).
13. S. Deering, "Host Extensions for IP Multicasting," *Internet Engineering Task Force,* RFC 1112 (August 1989).

Paul B. Patrick, Sr.

CASE Integration Using ACA Services

Digital uses the object-oriented software Application Control Architecture (ACA) Services to address the problems associated with data access, interapplication communication, and work flow in a distributed, multivendor CASE environment. The modeling of applications, data, and operations in ACA Services provides the foundation on which to build a CASE environment. ACA Services enables the seamless integration of CASE applications ranging from compilers to analysis and design tools. ACA Services is Digital's implementation of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) specification.

Based on work accomplished in many computer-aided software engineering (CASE) projects, this paper describes how Digital's object-oriented Application Control Architecture (ACA) Services can be used to construct a CASE environment. The paper begins with an overview of the types of CASE environments currently available. It describes the object-oriented technique of modeling applications, data, and operations and then proceeds to discuss design and implementation problems that might be encountered during the integration process. The paper concludes with a discussion of environment management.

CASE Environment Description

Today's CASE environments are required to operate in network environments that consist of geographically distributed hardware manufactured by multiple vendors. In such environments, access to data, metadata, and the functions that operate on this data must be as seamless as possible. This can be accomplished only when well-architected protocols exist for the exchange of information and control. These protocols need not be defined at the level of network packets, but rather as operations that have well-defined, platform-independent interfaces to predictable behaviors.

In addition to utilizing the various applications, environments deal with how applications are organized or grouped within a project and how work flows between applications and within the environment as a whole. These concepts are discussed later in the paper as are the different styles of integration that an application can employ.

Data integration, i.e., information sharing, is vital to any CASE environment because it reduces the amount of information users must enter. However, data integration must be accompanied by a mechanism that allows control to pass from one application to another. This mechanism, commonly called control integration, provides a means by which the appropriate application can be started and requested to perform an operation on a piece of information. Control integration is also used to exchange information between cooperating applications, regardless of their geographic locations. These two integration mechanisms used in tandem can solve many of the problems presented by a distributed, multivendor CASE environment.

ACA Services is Digital's implementation of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) specification. ACA Services is designed to solve problems associated with application interaction and remote data access in distributed, multivendor environments such as the CASE environments just described. This support includes the remote invocation of applications and components without the need for multiple logins or the use of terminal emulators. The encapsulation features of ACA Services allow the use of applications not designed for distributed environments. ACA Services can also be configured, in a way transparent to the application, for use on a local host.

The central focus of a CASE environment is on how easily functions such as compiling, building, and diagramming can be performed. The functions available form the foundation on which the environment is constructed. Therefore, the first step in the design of a CASE environment is to determine what functions to offer. The applications currently available to support these functions may be integrated using one of two paradigms: application-oriented or data-oriented.

Application-oriented Paradigm

CASE environments that follow the application-oriented paradigm focus on standalone applications used to develop software such as editors, compilers, and version managers. Application-oriented environments normally comprise a collection of applications that support the necessary functions. In application-oriented environments, integration tends to be focused on direct communication between two different applications. In this paradigm, the requesting application knows which class of application can be used to satisfy a particular request. Environments that present an application-oriented paradigm to the user require the user to have knowledge of the applications that can be used to perform specific tasks.

As the level of task complexity increases, it becomes increasingly important to build environments that utilize a paradigm focused on the data associated with the task being done and not on the applications used to perform the task. The realization of this problem has brought about the existence of data-centered environments.

Data-oriented Paradigm

CASE environments that use a data-oriented paradigm are centered around the data associated with the task the user is performing. To accomplish a task in such environments, operations are performed on a data object. Using the object being addressed, the operation, and preferences supplied by the user, the environment determines which application will be used to perform the requested operation. Thus, the requesting application requires no knowledge about which application implements an operation. This paradigm is extremely useful in CASE environments because of the diversity of objects and range of applications available to perform certain operations.

The application and the data paradigms can coexist in a single CASE environment, and in fact, tightly integrated CASE environments exploit the strengths of each paradigm. A text editor can be used to illustrate this point. Typically, when the contents of a source file need to be modified, an edit operation is sent to the object representing the file. However, a debugger may also use the same editor to display source code. The operation to position the cursor on a particular line is sent directly to the text editor application, rather than to a data object such as the line. An environment with such a split focus avoids the expense and complexity of presenting a complete object-oriented interface to the environment and results in the existence of both application- and data-oriented paradigms.

Regardless of which paradigms and applications a CASE environment uses, the primary focus of the environment is on the objects and on the operations that are defined on those objects. Therefore, after determining what functions to offer, the second step in designing a CASE environment is to understand how applications, data, and operations are modeled using an object-oriented approach, in particular the one provided by ACA Services.

CASE Integration in Object-oriented Terms

Describing environments using object-oriented techniques can simplify the design of an environment. Techniques such as abstraction and polymorphism can be used to describe the objects that comprise the environment, the operations that can be performed on those objects, and any relationships that exist between objects. Furthermore, using these techniques makes it possible to describe an environment as a set of classes and services for each class. ACA Services performs the role of the method dispatcher, matching an object and an operation with the function in an application that can implement that operation. To realize the benefits of this approach requires constructing models for the applications, data, and operations that will be present in the environment.

Modeling Applications and Application Relationships

Applications that are integrated into an environment can provide various functions or services to other members of the environment. The number of services an application provides depends not only on the capabilities of the application but also on the way it is modeled. These services are standalone pieces that can be plugged into a system to perform specific functions. An application can define a single operation whose sole function is to start the application; an application can export the entry points of its callable interface; or an application can define sets of operations for each type of object it manipulates. In support of application modeling, ACA Services provides the concepts of application classes, methods, and method servers. Figure 1 illustrates the relationships among the various pieces of information used to model an application in ACA Services.[1]

In ACA Services, the definition of an application is divided into two pieces: interface and implementation. The interface definition is concerned with the publicly visible aspects of the application. These include class definitions for the objects that the application manipulates, a class definition for the application itself, and definitions of operations that the application supports. The operations, which represent the functions provided by the application, are modeled as messages on the application class definition. These messages define a consistent interface to various implementations of the operations. Placement of the application class definition affects the behaviors this definition inherits. This is sometimes called classification. The classification of each component of an application depends on whether a component contains a superset or a subset of the functions contained in the components of other applications in the environment.

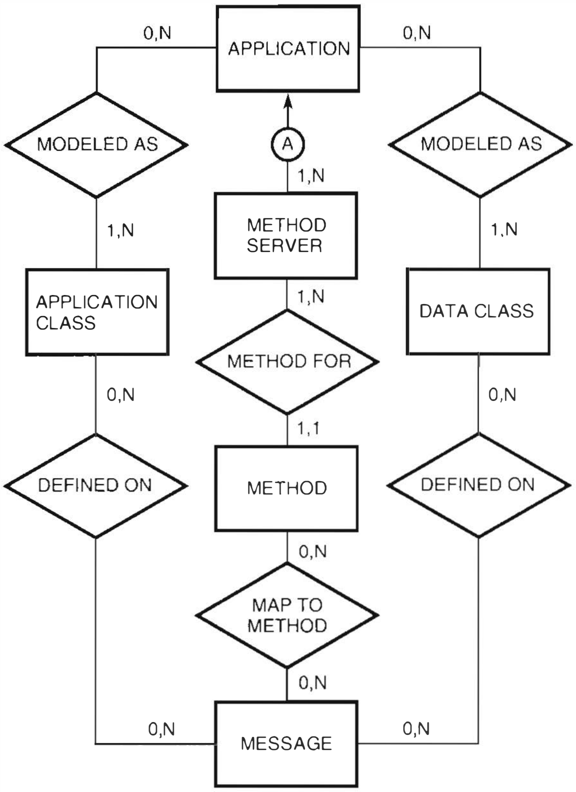


Figure 1 ACA Services Metadata Model

Once the application's components have been classified, the integrator must determine how the application will make its capabilities available to the environment: as an operating system script, as a callable interface, or as an executable image. The implementation definition represents the actual implementation of the application. An application may comprise a number of executable files and shared libraries. Typically, only the executable file used to start the application is modeled as a method server. If the functions of the application are provided through a shared library or image, only the shared library is modeled as a method server.

The implementation of the functions or services exported to the environment are modeled as methods. Methods describe the callable interfaces or operating system scripts that implement a particular operation and are associated with only one method server.[2] During the method selection process, the messages defined for the application and the objects it manipulates are mapped onto one or more methods.

Modeling Data and Data Relationships

Data modeling is another significant aspect of creating CASE environments, especially environments that utilize a data-oriented paradigm. Identifying the data objects that the application uses is a key element in the process of integrating that application. The list of data objects should include those objects for which the application provides a service, as well as those objects on which the application makes requests. The variety and quantity of data objects can vary from application to application and depend on an application's capabilities and the paradigm utilized. To support the modeling of data objects, ACA Services uses the concept of data classes. Note that, rather than provide instance management for data objects, ACA Services provides a means to represent the data classes used by an application as metadata.

Because environments that utilize a data-oriented paradigm may contain many data classes, ACA Services organizes the data classes into an inheritance hierarchy. This hierarchy allows responsibilities, such as operations and attributes, to be inherited by other data classes. Data classes found in an ACA Services inheritance hierarchy are related to one another through an "is-kind-of" relationship. A class that has an "is-kind-of" relationship with one or more superclasses must support all operations defined on the superclasses from which it inherits.[3] A subclass is not limited to those operations and attributes defined by a superclass but may have other operations, as well as refinements to inherited operations and attributes.

Modeling Operations

As mentioned previously, operations are modeled as messages in the CASE environment. The name of the message describes the type of operation. Some messages are data oriented, i.e., Edit, Reserve, and Copy, whereas other messages are application oriented, i.e., ExecuteCommand and TerminateServer. Messages provide a consistent abstraction of the functions provided by applications. This abstraction allows the details of how a function is implemented to be hidden from the requesting application. Since ACA Services supports more than one implementation for a single message, it also provides a means to hide various implementations.

The developer should anticipate different implementations of a message within the environment and be aware that a message may apply to a variety of classes. The developer must consider how the operation on an object might be used by various applications and in future environments.[4] In this way, adding new types of objects to an environment requires only minor changes, if any, to applications that are already integrated.

Operation Interactions The semantics of a message dictates which particular interaction model is to be used. ACA Services can be used to construct a number of different interaction models: synchronous request, asynchronous request, and request/reply, as shown in Figure 2. The synchronous request interaction model, shown in Figure 2a, is useful when serial operations originate from a single source. This model blocks the execution of the client application during a request. Control is returned to the client application only after the server application receives and executes the request and outputs data, if any.

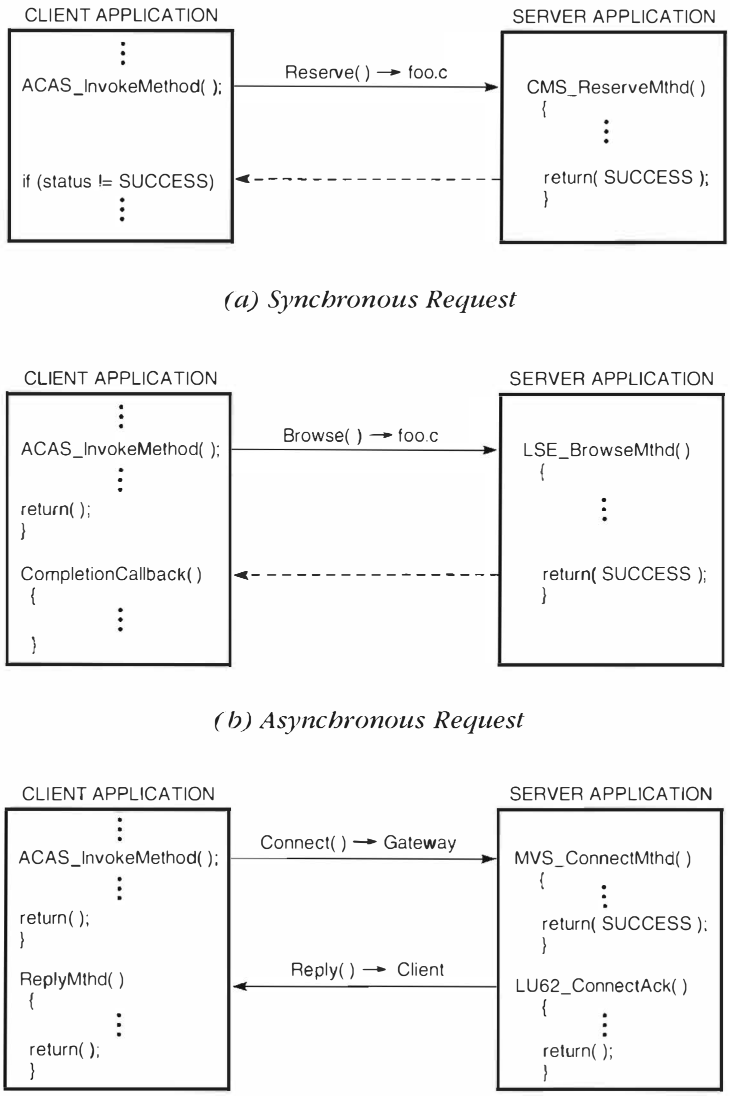
The asynchronous request interaction model, shown in Figure 2b, is useful in situations where the client can process other work until the server application completes the request. This model is especially beneficial when the requested operation takes a considerable amount of time to complete or if the server is busy with other requests. Execution of the client application is blocked only for the amount of time required to deliver the request. Client execution resumes once the request has been delivered. Upon completing the processing of the request, the server application notifies the client application of the completion and returns any output data.

The request/reply interaction model, shown in Figure 2c, is most appropriate for requests whose implementations cannot perform the operations required to obtain the necessary output data. Gateway and message-forwarding applications are examples of applications for which this type of interaction model is best suited. In this model, the message that represents the request cannot have any output arguments and must pass an application handle to itself. The server application uses the application handle to return any output information to the requester by sending a message that represents the reply. In a request/reply model, a single reply message should be defined for returning information, thus reducing the number of messages an application must support.

Message Arguments A message argument for passing the object being manipulated need not be defined. ACA Services automatically passes the object to which the message was sent to the method. Each method routine can access the object through a structure containing context information for the current invocation.

The arguments of a message should not be designed around a specific instance of an application, nor should they imply how an object is physically stored. To help meet these design criteria, all references to an object should be passed as instance handles. In this way, the application that receives the instance reference can use it directly for subsequent operations on that object. In addition, when defining the message arguments, developers should consider other applications that could be instances of a particular class and possibly used as replacements.

However, all instances of an application do not have the same set of capabilities. To support the various capabilities, the developer may have to define additional arguments to represent bit masks and flags. An argument list or an item list can be used to pass information about different data types or quantities. The message design should not require implementation-specific information for proper application operation; this design implies that reasonable defaults accommodate any unspecified



(a) Synchronous Request

(b) Asynchronous Request

(c) Request/Reply

Figure 2 Operation Interaction Models

information. In cases where proper operation of an application requires implementation-specific information, the most suitable design is to use the context object as a place to store the default values. With such a design, the application no longer needs to use hard-coded default values and can be customized for the environment.

Integration Frameworks

A number of issues must be resolved in the construction of a CASE environment before the first line of code can be written. Many of these issues center around the modeling of objects in the environment. As discussed in the previous section, abstraction is used to hide much of the actual implementation of the operations on objects from the requesting application. However, additional context may be required for further operations. If the application is using an application-oriented paradigm, most operations are directed to an application class that provides the service. In cases where a data-oriented paradigm is used, the application typically directs operations to the data class of which the object is an instance.

Besides the application and data objects found in the environment, the designer must also take into consideration the other components of the CASE environment itself. Figure 3 shows the major components of a CASE environment: activities, applications, application and data interfaces, work flow management, and handle management. Each component represents a particular aspect of the overall environment. The components are introduced in this section and described in detail elsewhere in the paper, as indicated.

Activities provide the basic work structure for a particular task within an environment. Each activity comprises one or more applications and a number of data objects, forming a single composite object. Applications within an activity operate through the application interfaces. The section Application Integration describes the principles of an activity and includes a discussion of the sharing of applications within and among other activities.

Application interfaces, illustrated in Figure 3 as arrows connecting the various applications, form the primitives by which integration is accomplished. Some of the more general concepts for application interfaces were discussed in the section Modeling Operations; these concepts are described in detail in the section Styles of Application Interfacing.

Finally, the section Environment Management addresses how to manage the flow of work within the environment. This section describes the management of instance and application handles, the use of storage classes as a means to provide data transformations, and the management of events within the environment. To better understand each of these topics requires the following basic information about various aspects of the environment.

Adding New Implementations

Updates to the environment may include adding new application classes, data classes that the new application supports, method definitions for the application, and possibly a method server definition. As described earlier in the paper, ACA Services uses data and application classes to represent the different classifications of data and application objects found in an environment. Storage classes represent the classifications of storage and how objects are referenced in the environment. Each class, i.e., data, application, and storage, contains a list of messages that represent the operations that can be performed on the class.

Digital's CASE environment, COHESION, was designed to present a data-oriented perspective to the user. An initial level of integration was achieved by utilizing this same data-oriented approach to application integration. Implementation of a data-oriented approach required that method maps for messages on data classes contain an indirect reference to an abstract application class.[5] Figure 4 illustrates this concept by showing two different messages: the Edit message, which uses an indirect method reference, and the Browse message, which uses a direct method reference. An indirect method reference has two parts separated by the character '@': first, the name of the message to be sent; and second, the name of the class on which to send the message. Although not commonly done, an indirect method reference allows the original message to be mapped to another message on a different class, given that both messages have arguments of the

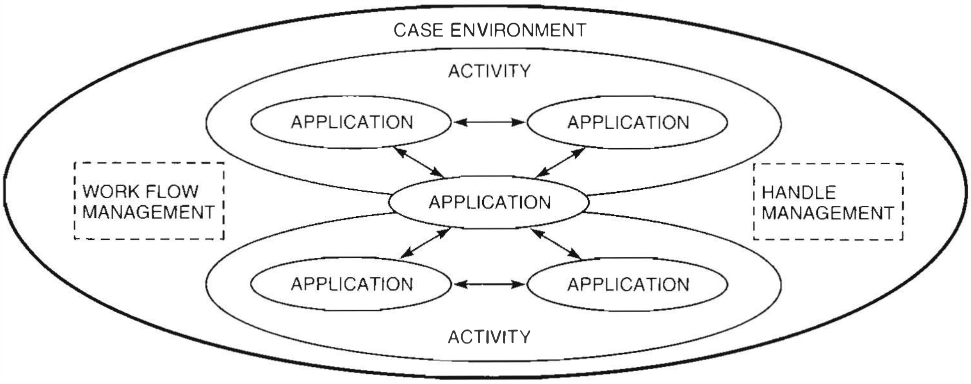


Figure 3 Components of a CASE Environment

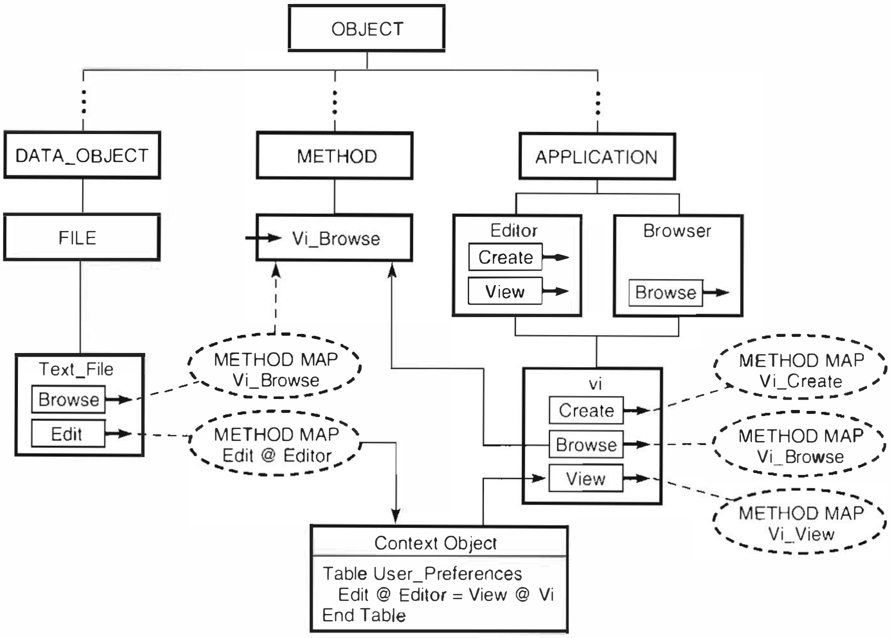


Figure 4 Direct and Indirect Method References

same type, direction, and order. Both messages must also return the same type of object.

On encountering an indirect method reference, ACA Services first looks at tables in the context object for an attribute that matches the reference. If such an attribute is found, ACA Services uses the attribute value to determine the class and message that should be checked next. Thus, users can provide a mapping to their preferred application for the operation. If no matching attribute is found, ACA Services uses the message and class specified in the indirect method reference as the next place to check.

The approach used in COHESION has many advantages over specifying either a direct reference to a method or an indirect reference to a specific application class. This approach does not limit the user's ability to specify application preferences associated with using direct references to methods, nor does it burden the installation of the application with determining all the data classes that will need to be updated (as required with indirect references to a specific application class). In addition, the approach allows the application developer to do the least amount of work and still provide the maximum level of support for user preferences in applications.

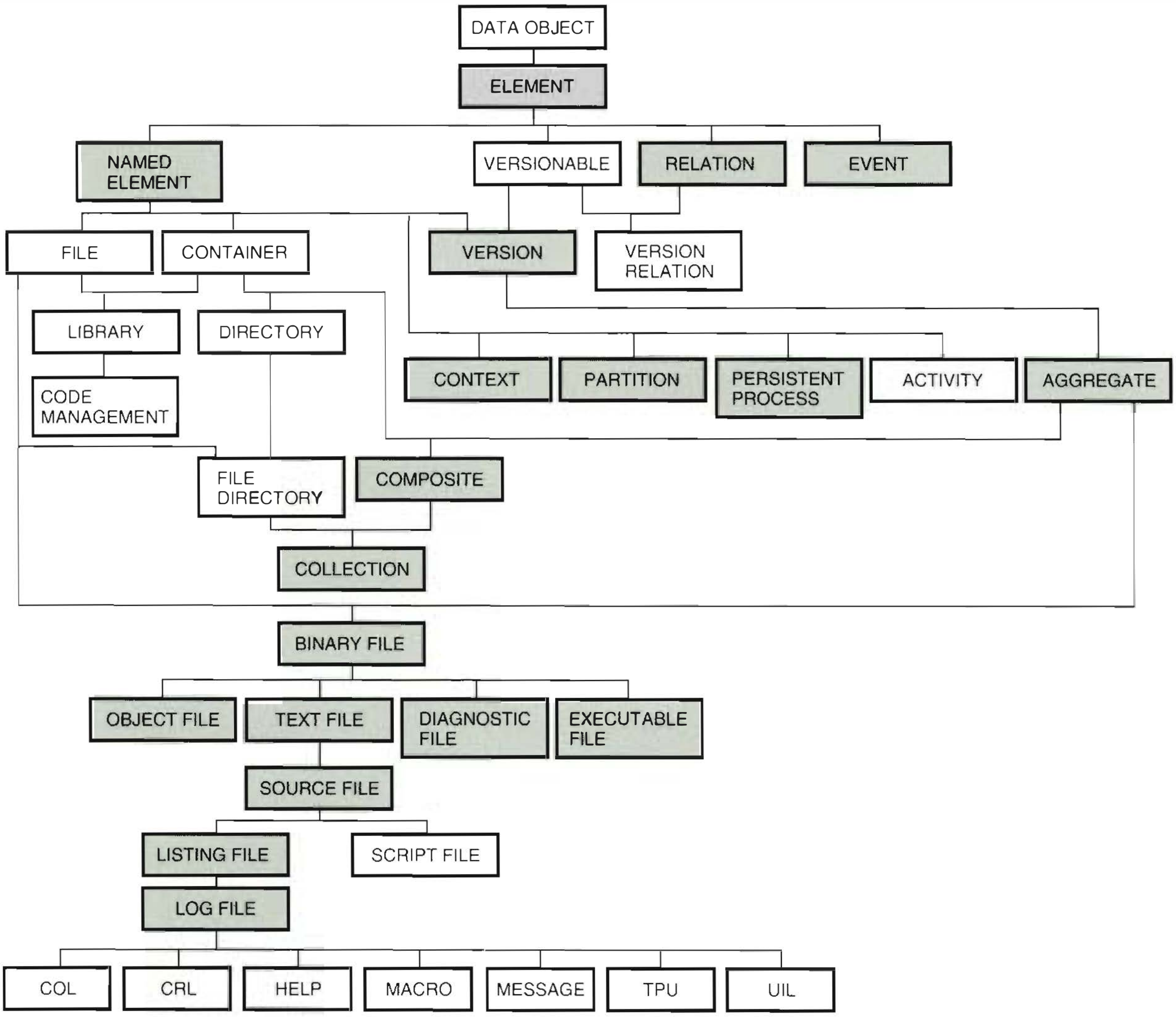
Using ACA Services, the application developer must create an application class definition for each CASE application to be added. Consequently, the class hierarchy contains both abstract and instance classes. The application class is required to contain all the messages defined on its superclass, plus any additional messages that the application supports. The method map of each message on an application class should contain a direct reference to the method that implements the operation. Although better than the other alternatives, the COHESION approach has no default implementation unless one is explicitly specified in a context object. To overcome this problem, an entry for each message defined on the abstract application class must be created in one of the context objects. The values for these entries point to the corresponding message on the class of application used as the default implementation.

Common Classes

Common classes for a CASE environment provide CASE application developers with a description about how an application fits into the environment, the behaviors the application must support, and the messages that result in those behaviors. The notion of plug-and-play in the environment is achieved through the use of common classes. An implementation that adheres to the description of a particular class of applications can be easily switched with another implementation that adheres to the same application class semantics.

Programs like COHESION are working toward a set of common classes for CASE environments. The set currently defined contains classes for many types of data and applications found in CASE environments focused on the coding and testing phases of the software development process. A graphical view of the data portion of the hierarchy is shown in Figure 5. The hierarchy is partially based on the hierarchy found in ATIS, a standard for tool integration, and utilizes the strength of the ATIS data model.[6] (Shaded boxes indicate the classes that are specific to ATIS.) Encompassing the ATIS model, the hierarchy presents a uniform data model for the integration of data throughout the CASE environment. The set of classes, although not exhaustive, serves as a basis on which a CASE environment can be built. Extensions of the hierarchy will occur as new classes of applications and their associated data objects are integrated into the environment by independent software vendors, customers, and other CASE vendors.

Most data classes are subclasses of the data class SOURCE_FILE, because the initial data class implementation was targeted at a CASE environment consisting of editors, compilers, builders, and analyzers. Additional data classes for both file and nonfile objects will be added when applications that provide and manipulate these objects are



Note: Shaded boxes indicate ATIS-specific classes.

Figure 5 Hierarchy of CASE Common Data Classes

integrated into the environment. A number of data classes represent composite objects such as tests and activities. These data classes are used to hide how the object is physically stored in the environment. Classes that represent composite objects have attributes with values that are actually other objects. For example, the test data class typically has attributes that represent the result of a test run, an operating system script or program used to perform the test, and a benchmark against which a test run is compared. Each of these attributes may have as a value a reference to the file object that contains the actual data.

The portion of the hierarchy that is used to specify application classes contains only abstract application classes, as shown in Figure 6. These classes provide structure, but more important, they define the operations that are inherited by any application that is an instance of a class. Abstract classes are provided for a number of the applications found in CASE environments that deal with the coding and testing functions. The hierarchy does not contain any classes that represent particular instances of an application. Such application classes exist only when applications are installed in the environment.

Consistent Integration Interface

Many CASE vendors are building products for a number of different environments, including electronic publishing, office automation, computer-aided design, and computer-aided manufacturing, in addition to CASE. Therefore, vendors must decide how to integrate these applications into the various environments. Until now, most integration was accomplished by linking one application with another, which resulted in tightly coupled applications. However, such applications tend to be unable to operate independently, without the other member. Also, each coupled member tends to have its own application programming interface (API). Integration performed in this manner results in an application that must maintain code to support multiple APIs, if the application is to work in a number of environments. Such support can increase the maintenance cost and the time and effort required to integrate with other implementations of applications and environments. Other by-products of this approach are an increased image size and a need to rerelease software when a dependent application changes. The degree to which rerelease occurs varies with the platform and operating system.

ACA Services can be used to minimize the number of interfaces that an application must maintain without removing functionality; a common API provides the interface to all potential functionality. The ACA Services API, along with a set of common classes, allows the same level of interaction between applications that can be accomplished through a private API, without the negative side effects previously described. Through the use of common classes, an application can integrate with multiple implementations of another application without requiring a separate effort for each. On platforms where dynamic loading of libraries or shareable images are supported, applications can use ACA Services to locate the appropriate library,

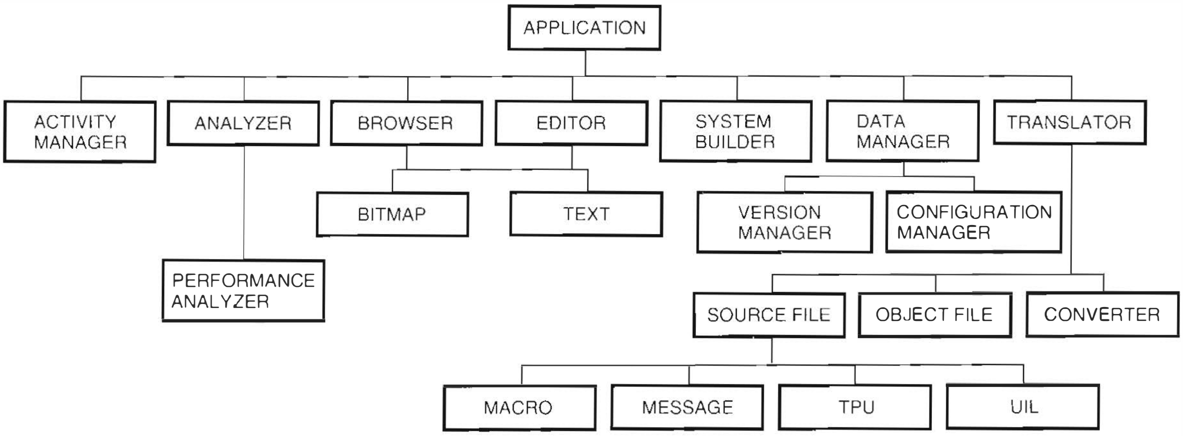


Figure 6 Hierarchy of CASE Common Application Classes

find the proper entry point, and transfer control to the appropriate routine. ACA Services also provides a transparent mechanism for encapsulating applications that have no callable interfaces. Use of this mechanism extends the number of applications that can be integrated and removes the need to develop operating system-specific code to start applications.

Styles of Application Interfacing

Creating an interface to an application that is to be integrated is different from integrating an application into an environment. Application interfacing deals with the public interface or interfaces that the application provides to another application. In turn, these interfaces provide the primitives that can be used in the integration of applications.

Application interfaces can be created in various ways, with differing levels of effort. Software developers can design new applications to utilize all the capabilities of ACA Services. Existing applications can also take advantage of the full capability of ACA Services, if the source code to the application is available and if the application can be easily adapted to use an event-driven model. However, even if the source code to an application is not available, applications can still be integrated into the environment using ACA Services. If the application has a callable interface, a server can be written that receives messages and calls the appropriate API routines. If the application does not have a callable interface, the application can be integrated by encapsulation through the use of an operating system script. The remainder of this section describes how to use each of these techniques to create an interface through which the application can be integrated into a CASE environment.

Application Modifications

An existing application can easily be adapted to use ACA Services, if the source code to the application is available. With minimal changes, an application that utilizes an event-driven design, like that used by most window-based applications, can operate as an application server. The actual modifications required to provide ACA Services support differ across applications, but for most window-based applications the changes are similar. As an illustration of this style of integration, consider an editor.

Most editors are implemented as event-driven applications, which allows easy integration because the structure of the code requires no major changes. To register the current executing instance of the application with ACA Services, a call to the ACAS_RegisterServer routine must be added to the application's initialization routine. During the process of run-time registration, ACA Services registers various inf•rmation about the application, including the identifier of the process in which the application is executing, the owner of the process, and the class- and instance-unique identifiers for the application. As part of the registration, an application can specify an abstract name by which it can be located and the routines to be called when an ACA Services event arrives, e.g., when the server is instructed to shut down or when a session ends.

Once registered with ACA Services, the application must enter its event dispatching loop. Because many applications have existing event dispatching mechanisms, ACA Services has been designed for easy integration with most mechanisms. ACA Services provides this support by allowing the application to define a routine called the event notifier, which is called at signal level each time an ACA Services event occurs. The event notifier routine places an event on the applications work queue for the ACA Services event. Upon encountering the event, the application's event dispatcher routine calls the ACAS_Dispatch routine to allow ACA Services to dispatch the appropriate method or management routine for the event. A description of how ACA Services dispatches operation requests follows.

Application Servers

When the application to be integrated does not have a user interface but provides a callable interface, integration is best accomplished by creating an application server. Considered a form of encapsulation, an application server provides a consistent programming interface to the application. An application server provides jacket routines that use the application's callable interface, hiding the actual details of this interface. This technique is also used to create applications that have a clean separation of presentation and functions.

Applications that implement persistent data stores, such as databases, code managers, and repositories, are prime candidates for this style of integration. By using an application server to access persistent data stores, a requesting application need not know how the data store is implemented and which implementation is to be used. This technique promotes the reuse of existing functions contained in the environment regardless of the actual implementation of the function. Digital's Code Management System (DEC/CMS) and CDD/Repository software are examples of applications that have been integrated using the application server technique. Figure 7 illustrates the typical structure of the various components involved in this style of integration.

As shown in Figure 7, the integration process involves the following steps. (1) An invoke from the client application of the message "Reserve" on the object "foo.c" goes through the resolution code and (2) out the transport to the server application. This may result in starting the server application, if no server was available to service the request. (3) The server application's main routine calls the event dispatcher and waits for work to arrive, when the server is started. (4) When the "Reserve" message arrives on the transport, the transport notifies the server application, (5) causing the event dispatcher to dispatch the "Reserve" message by calling the method dispatcher routine. (6) The method dispatcher routine calls the appropriate method interface routine. (7) The method interface routine does any work required to call the appropriate callable interface routine. (8) When the callable interface routine returns control to the method interface routine, the routine can perform any work necessary before (9) returning control to the method dispatcher routine. (10) The method dispatcher routine then puts any arguments to be returned in the proper format and sends this information to the transport, which actually sends the information back to the client application.

Using the DEC/CMS application server as an example, the software developer must create a main routine to (1) perform any setup required to use the callable interface and (2) register the existence of the server with ACA Services. Registration includes specifying the method dispatcher routine, which is generated by ACA Services, so that the appropriate method routine will be dispatched for the message received.

A method routine exists for each operation that the server is capable of performing. The set of method routines is analogous to the operating system script for compilation used to explain application encapsulation later in this section. Because the DEC/CMS application server is not an operating system script, message arguments are passed into the method routine directly. As mentioned earlier in the section CASE Integration in Object-oriented Terms, the object on which the current operation is to be performed is available to the method routine through the use of the invocation context structure. Information about the object, such as its class, name, and generation, can be obtained by calling the ACAS_ParseInstanceHandle routine. The class of the object can then be used to determine if the object is an element under version control, a collection, or a group.

The name of the object and its generation are contained in the reference data field of the instance handle that represents the object. Because each

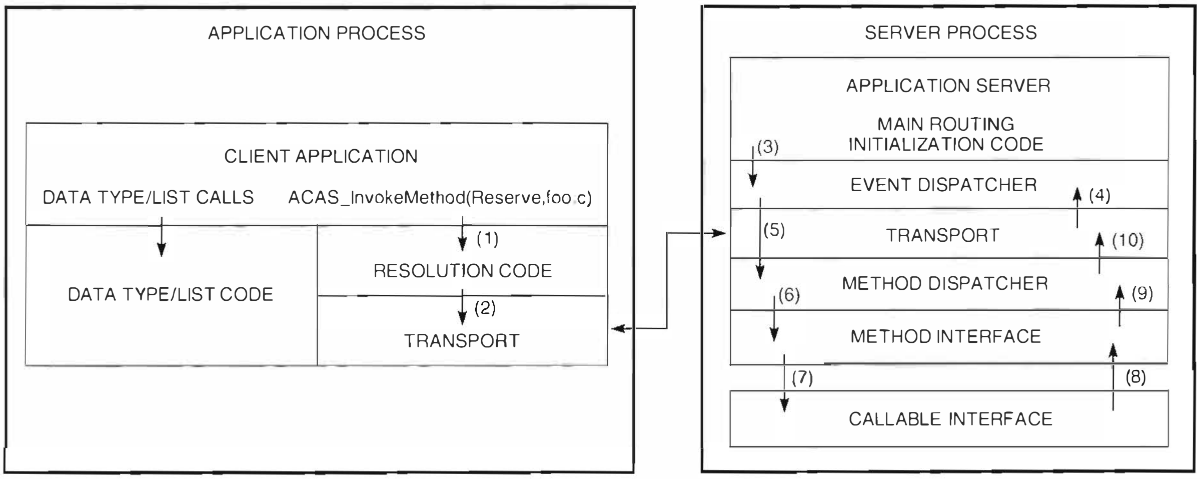


Figure 7 Block Diagram of a Code Management System Application Server

different code management system has its own representation of generation, it was necessary to create a canonical format to represent all implementations. Therefore, the method must convert the canonical generation representation to a format that is native to the implementation, i.e., DEC/CMS specific. In addition, any method that returns a reference to a versioned object must convert the native generation representation to its canonical format. Table 1 shows how an object reference can be mapped between its canonical and DEC/CMS-specific formats.

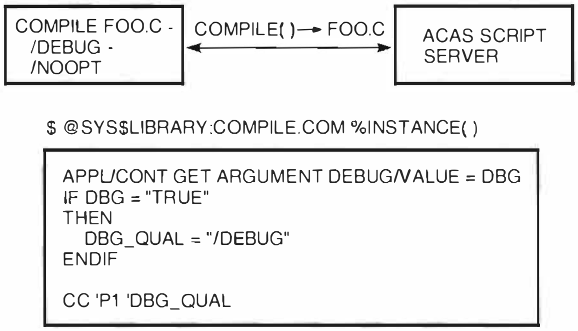
Once the necessary information about the object has been retrieved and converted to a format native to the implementation, the method can call to the appropriate callable interface routine, possibly based on the object's data class. Once the call completes, the method must convert any objects to be returned into a canonical format, at which point the method can return the status of the operation and output arguments.

Application Encapsulation

Encapsulation, the simplest integration technique, is appropriate for applications that do not have a callable interface or in cases where no source code is available. Compilers are an ideal candidate for this style of integration, because they perform synchronous operations. Encapsulation of compilers provides a consistent programming interface to any compiler that is integrated into the environment, regardless of the qualifiers used to specify particular compilation options. This technique can also be used to provide a generic compile command that is platform independent. Encapsulation of a compiler is best accomplished through the use of an operating system script. Figure 8 illustrates an example of an encapsulated compiler.

Table 1 Converting Generation Representations

	Native Representation	
Canonical Format	**Object Name**	**Object Generation**
UTIL.C(10:BL7:3)	UTIL.C	10B3
DISPATCH.C(1)	DISPATCH.C	1
DUMP.B32(1:A:8)	DUMP.B32	1A8
GRAPH.BAS	GRAPH.BAS	1+



```
$ @SYS$LIBRARY:COMPILE.COM %INSTANCE( )

APPL/CONT GET ARGUMENT DEBUG/VALUE = DBG
IF DBG = "TRUE"
THEN
   DBG_QUAL = "/DEBUG"
ENDIF

CC 'P1 'DBG_QUAL
```

Figure 8 Example of an Encapsulated Compiler

The purpose of an operating system script for compilation is to convert the generic compilation qualifiers, which are passed as message arguments, into the compiler-specific options. The /DEBUG and /NOOPT qualifiers shown in Figure 8 are examples of generic compilation qualifiers. Many operating system scripting languages limit the number of parameters that can be passed on the command line. The compilation scripts avoid these limitations by passing the name of the file to be compiled as the only command line parameter, as shown in the command @SYS$LIBRARY:COMPILE.COM %INSTANCE( ) in Figure 8. ACA convenience commands, such as APPL/CONT GET ARGUMENT, are used to retrieve and set the values of the message arguments in the operating system script. When all the switch values are gathered, the operating system script converts the generic values into specific qualifiers. Finally, the actual command line is constructed and executed. This same technique can also be used to encapsulate linkers and any other types of applications where no source code or callable interface is available. When applications provide a callable interface, even tighter integration can be achieved by creating an application server.

Application Integration

Integration of applications goes beyond the interfaces that applications present to the environment; it concerns how applications interact with one another. Integration also takes into account the policies used in an environment to allow a collection of applications to be grouped into a single composite object. This section discusses concepts such as an activity, locating an application within an activity, context sharing, and the sharing of applications across multiple activities.

Activity Participation

Since more than one activity may be active at any given time, an activity must be able to locate the other applications participating in the activity. Data-oriented environments provide a means to loosely couple the various data and application objects into a single composite object. The COHESION integrated environment refers to this composite object as an activity. The implementation of an activity differs depending upon the environment: ATIS uses a persistent process; file system-based environments generally use a directory hierarchy; and environments built on a private data store can use a data file. In the COHESION environment, an activity is represented as an ACA Services context object that contains attributes that reference a directory hierarchy. The context object is used to set up the execution environment in which a set of applications will operate and to locate other applications that are executing within the activity.

Locating Activity Applications

The ability to locate an application that is executing in an activity allows for reuse of the application by other applications executing in that same activity. Such locating provides for better utilization of applications and reduces the amount of context that must be propagated from one application to another. To locate an application within an activity, an application must have registered its presence in the activity. When registering with ACA Services, the application must specify the activity name as the value of the attribute ACAS_SERVER_REGISTRY. The application must also register itself with the event manager to allow centralized management of the activity and to participate in the flow of work within the activity.

CASE applications determine if they are executing within an activity by checking for the existence of the environment variable ACTIVITY_NAME. If this environment variable exists, its value is the activity identifier. To allow an activity to extend beyond a single host and to support different activities with the same name, the activity is identified by a unique identifier.

Sharing within Activities

Applications executing within an activity operate in a common context. ACA Services provides a set of mechanisms that can be used to provide this common context. The environment variable ACTIVITY_NAME is defined each time a method server is started in the COHESION environment. The method server definition specifies as the value of the start-up environment attribute, the names of the context tables and attributes that are to be defined as environment variables upon start-up.

Another way of providing a common context across an activity is to propagate context object tables and attributes as implicit arguments to method servers. Specifying this information as implicit arguments instructs ACA Services to propagate these attributes to the context object of the method server servicing the request.

The context object can also be used directly to create a common context across an activity, i.e., by holding information that needs to be shared. This information can include references to directories, preferences of applications, and default values.

Sharing between Activities

Reusing applications that are active within an activity reduces the overall system resources required to perform the activity. However, a problem occurs when two or more activities are active at the same time and require the same application. With the addition of windowed interfaces and the need to utilize other services, application sizes have greatly increased. Consequently, it is often impractical to expect a separate instance of an application to be associated with each activity that is active.

In order for an application to be shared between multiple activities, the application needs a means by which to determine if a request is part of an ongoing dialog with another application or is the beginning of a new dialog. These dialogs, called "sessions," represent a conversation between a pair of applications. Each time a client application makes a request to a new application server, a session is established and an identifier is associated with the session. ACA Services passes the session identifier to the server application.

The management of sessions can be accomplished by using the session ID as a lookup key into a list of structures that represent the active sessions. When the server application locates the structure associated with the session identifier, the application can establish the appropriate context for that session. In the example of DEC/CMS application server, the structure would contain the handle to the library associated with the session.

ACA Services also notifies an application server when a session is to be terminated between a client and a server application. When notified, the application server determines the appropriate course of action. Using the CMS example, the server releases any cached information it has kept about the session, closes the specific CMS library, and then frees the library data block.

Environment Management

After defining application interfaces and integrating applications into an activity, CASE environment developers must focus on the management of the environment as a whole. This includes the management of references to applications and data, the transformation of object references into platform-specific formats, and the flow of work within the environment.

Handle Management

In the CASE environment, objects are the targets of all operations. Sending a message to an object requires understanding how to create and manage references to the object. Since ACA Services does not manage instances of objects, it uses references to instances of objects. These references take the form of instance and application handles, which reference data and application objects, respectively. Proper management of these handles leads to more efficient use of application objects, thus reducing the amount of network resources and memory consumed by the application. Appropriate handle management can also enhance performance and guarantee predictable behavior.

Instance Handles

The creation of an object reference is performed by calling the ACAS_CreateInstanceHandle routine. ACA Services (1) creates an instance handle from the information passed as arguments to the routine, (2) allocates memory to the handle and manages this memory, and (3) sends a message to a storage class, if one was specified.

To avoid creating numerous copies of an instance handle, each with its own memory, a cache of objects should be used. This is especially true in CASE environments that use the data-oriented paradigm. Each object structure contains pointers to both the previous and the next object structure in the queue. The structure also contains values for the location and reference data fields that were passed as arguments to the ACAS_CreateInstanceHandle routine and, thus, allows for the unique identification of an object in the cache across multiple hosts. In addition to the location and reference data, the structure contains a pointer to the instance handle returned from the call to the ACAS_CreateInstanceHandle routine. Reuse of the instance handle saves the time required to create the handle, including any overhead associated with using storage classes. Reuse also reduces the total amount of memory required. However, instance handles are not the only handles that require management; application handles need to be managed as well.

Application Handles

Application handles are references to application objects. Each application handle can represent one or more method servers. A method server can generate a handle by calling the ACAS_CreateApplicationHandle routine, or the ACAS_InvokeMethod routine can return an application handle as an output argument. As with instance handles, application handles can be passed as arguments to a message. Management of application handles is similar to the management of instance handles. Each entry in the cache of application handles contains the location of the application and the name of the class of application. The entry also contains a pointer to the application handle and a count of the number of outstanding references to the handle. Freeing an application handle results in the termination of all sessions between the client and any method servers referenced by the handle; it also releases all memory associated with the handle.

Each instance handle should be associated with a corresponding application handle. This association allows the application handle to be reused when sending additional requests to the application concerning the data object. An application handle associated with a cache entry can be used to make the request. Failure to find the application in the cache could indicate that the appropriate invocation flag should be used to obtain an application when calling the ACAS_InvokeMethod routine.

As described, proper handle management can result in better performance, better resource utilization, and predictable behavior within the environment. However, handle management does not deal with how to create an object reference that, when presented to an application on a remote host, is in a format native to that platform. For this capability, we must turn to storage classes.

Data Transformations Using Storage Classes

Distributed CASE environments, whether homogeneous or heterogeneous, must concern themselves with the representation of object references that are shared among different applications. File specifications exemplify this problem. Given multiple hosts, it is unlikely that two hosts have the same path to a specified file, even if both hosts are of the same platform type. Consider the scenario in which Application A sends the Edit message to the file object $PROJ4:[PROJECT.SRC]SORT.C, resulting in a request of Application B to edit the contents of the file. The problem becomes complicated if Application B is executing on a different platform type than Application A.

To solve the problem, the environment can utilize the functionality provided by ACA Services storage classes. Storage classes provide a mechanism for translating an object's reference data from one file system representation to another. A solution to the scenario described involves implementing a set of methods that would be executed when the object reference uses a storage class.

The SC_COHESION storage class is a CASE-specific storage class, which is a refinement of the SC_FILE storage class provided by ACA Services. As a refinement, SC_COHESION inherits all the messages defined on its parent storage class, including the messages SetInstance and GetInstance. The methods for these two messages provide an implementation for mapping file system specifications from platform-specific formats to platform-independent formats and back again. The storage class methods do this by utilizing device and directory information, called directory mappings, found in the context object.

The directory mappings stored in the context object provide a means to associate a physically shared directory path with a network path name. The network path name is a platform-independent name that, when presented to a remote platform, can be mapped into a format native to the platform receiving the request. A network path name and its mapping are stored as an attribute-value pair in the PATHNAME_REGISTRY table of a context object.

The directory mapping functionality allows references to file objects to be passed between applications on different hosts in a way independent of the platform. This same scheme can also be used to convert object references in object identifiers, such as ATIS element IDs for use with the CDD/Repository software. In the implementation for the file system, the method associated with the SetInstance message must determine the data class of the object reference, as well as transform the reference data into its network format. The determination can be made in a number of ways, the most common of which is to base the class on the extension of the file. Although not the most accurate method of determining the class, this approach does meet the needs of many files.

Work Flow Management

ACA Services manages the various instances of executing applications but does not understand the concept of an activity. Therefore, managing the applications within the activity requires the use of an application that understands this concept. The event manager, which acts as a central registry of active applications and their associated activities, can provide a simple form of work flow management within the environment. However, the event manager is used only in a limited capacity in the COHESION integrated environment. In COHESION, the event manager is notified each time an application is started or stopped in an activity. The application provides an application handle to itself, which is used by the event manager to notify the application of events of interest. The use of the event manager removes the need for an application to forward certain messages, as a result of an event in the environment, to all applications with which it has been communicating. Removing the need to forward messages reduces both the chances of loops forming in a set of applications and any communication deadlocks between applications.

Events and Triggers

On registration, an application can express interest in being notified about particular events. Events are categorized into two classes: system events and application events. System events affect the overall operation of the environment. These events include shutdown and changes in activities. All applications in the COHESION environment are notified of the system events for activity shutdown, iconification, and deiconification. Application events occur when the state of an object in the environment changes. File modification or completion of a build step are typical examples of application events. Other applications in an activity can use these events for synchronization or as notifications that cause a change in behavior. Such notifications have traditionally been called triggers.

For example, in a simple build system such as the make utility, events can create a work flow that would automatically compile and link an application when one module changes. If the build process completes successfully, the work flow automatically starts the debugger to debug the newly built executable file. If the build fails, the work flow loads the faulty module into a program editor and positions the cursor to the line where the error occurred.

Summary

ACA Services can be used to resolve many problems encountered in a distributed, multivendor environment. The object-oriented approach provided by ACA Services can aid in the construction of a CASE environment that promotes the plug-and-play concept across a number of different platforms and network transports. ACA Services provides a means of developing client-server applications and of abstracting the network dependencies away from the developer. This feature, together with the use of storage classes and data marshaling, can help to exchange information in a heterogeneous environment. At the same time, ACA Services can provide a consistent programming interface to all components in the system. The dynamic nature of ACA Services allows new components to be added to the environment without the need to rebuild the entire environment. The flexibility of ACA Services allows its use to construct a CASE environment regardless of the integration paradigm used and while supporting a number of interaction models. ACA Services provides the infrastructure necessary to integrate the large number of existing applications into distributed, heterogeneous environments.

Acknowledgments

The author wishes to thank Jackie Kasputy, Chip Nylander, and Gayn Winters for their invaluable insights and contributions on distributed, multivendor CASE environments.

References

1. E. Yourdon, *Modern Structured Analysis* (Englewood Cliffs, NJ: Yourdon Press, 1989).

2. *DEC ACA Services System Integrator and Programmer's Guide* (Maynard, MA: Digital Equipment Corporation, Order No. AA-PQKMA-TE, 1992).

3. G. Booch, *Object Oriented Design with Applications* (Redwood City, CA: Benjamin/Cummings Publishing Company, 1991).

4. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software* (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990).

5. *DEC ACA Services Reference Manual* (Maynard, MA: Digital Equipment Corporation, Order No. AA-PQKLA-TE, 1992).

6. J. Liu, "Future Direction for Evolution of IRDS Services Interface," X3H4/92-161, Proposed specification submitted to ANSI X3H4 and ISO IRDS, 1992.

David Ascher |

DEC @aGlance—Integration of Desktop Tools and Manufacturing Process Information Systems

The DEC @aGlance architecture supports the integration of manufacturing process information systems with the analysis, scheduling, design, and management tools that are used to improve and manage production. DEC @aGlance software comprises a set of run-time libraries, an application development tool kit, and extensions to popular spreadsheet applications, all implemented with Digital's object-oriented Application Control Architecture (ACA) Services. The tool kit helps developers produce DEC @aGlance client and server applications that will interoperate with other independently developed DEC @aGlance applications. Spreadsheet extensions (add-ins) to Lotus 1-2-3 for Windows and to Microsoft Excel for Windows allow users to access real-time and historical data from DEC @aGlance servers. With DEC @aGlance software, control engineers and other manufacturing process professionals can use familiar desktop tools on a variety of platforms and have simple, interactive, and transparent access to current and past process data in their plants.

At a chemical plant that has been producing nylon using the same process for over 35 years, the lead control engineer told an interviewer that what he likes about his job is that "it is totally different every day."[1] To an outside observer, the operation of a process plant, such as a refinery or paper plant, appears to be an unchanging flow of materials into a tightly controlled and repetitive process that produces a continuous flow of unvarying product—24 hours a day, 365 days a year. In reality, the operation of these plants is far more complex and challenging, involving constant adjustment to changing conditions, aging equipment, and variations in raw materials, as well as constant monitoring for equipment malfunctions.

The operation of a large process plant involves the functioning of numerous valves, switches, pumps, other actuators, and sensors measuring and controlling the levels, pressures, temperatures, and flows of various materials through a complex series of pipes, tubes, tanks, and vessels. In addition to detecting and managing failures in these components, a large proportion of the personnel in the plant is involved in process and product improvement. The personal computer or workstation and an array of sophisticated desktop tools allow data to be analyzed, visualized, manipulated, and explored in ways that support creative problem solving. Getting timely information about the process into the appropriate problem-solving tools is, however, difficult. This paper begins with some background about manufacturing process information systems and the need for access to system data. The paper then describes the development of DEC @aGlance software and the choice and use of Application Control Architecture (ACA) Services to solve the problem of integrating independently developed applications in the manufacturing space.[2]

Background

In large manufacturing facilities, the production process is controlled through the use of advanced automation systems. These systems may track thousands of temperatures, flows, pressures, and levels and can drive hundreds of pumps, valves, and other actuators. To implement control strategies, such systems may compute large numbers of complex, dynamic control algorithms. Usually, additional systems measure various physical properties of the product, such as color, weight, viscosity, thickness, and moisture content. Supervisory control systems often coordinate parts of a complex process, as well as implement higher-level control and production strategies and keep historical records of key process variables.

The control of a large plant is usually implemented through strategies that allow the control problem to be divided into smaller parts, as illustrated in Figure 1. Each piece of the system is responsible for the control of a subsystem (e.g., steam generation and distribution, or cooling fluids), a part of the process (e.g., premixing, material storage, or reaction), or an area of the plant (e.g., packaging line, product stream, or finished goods management). Within each subsystem, there is typically a hierarchy of control. The lowest-level components control activities that require responses within less than a second to as much as one minute (direct control). The next level of systems control activities that require responses within less than a few minutes (distributed control). Above this level of response are systems that control activities that may not change for long periods or that implement control algorithms that involve measurements from more than one lower-level system (supervisory control). At the plant level, additional control systems may exist to implement control algorithms that reflect changes in the markets for products, market opportunities, and fluctuations in raw material availability and composition, along with the information about the process that is supplied by the lower-level systems (high-level control). Scattered among these levels may be various additional systems that schedule preventive maintenance, identify equipment failures, and advise on process improvements—all based on information about process from the other systems in the plant.

Distributed control systems include an operator console that consists of multicolor displays, push buttons, warning lights and buzzers, a touch screen or trackball, and industrialized keyboards with as many as a 36 special function keys. The displays allow an operator to oversee all parts of the process for which the operator is responsible. Typical displays show recent trends of key variables and mimic diagrams showing the current state of the manufacturing equipment (e.g., valve positions and tank levels) and of the material flowing through the process. The keyboard and other input devices allow the operator to select displays, request reports, and modify control settings. Response to

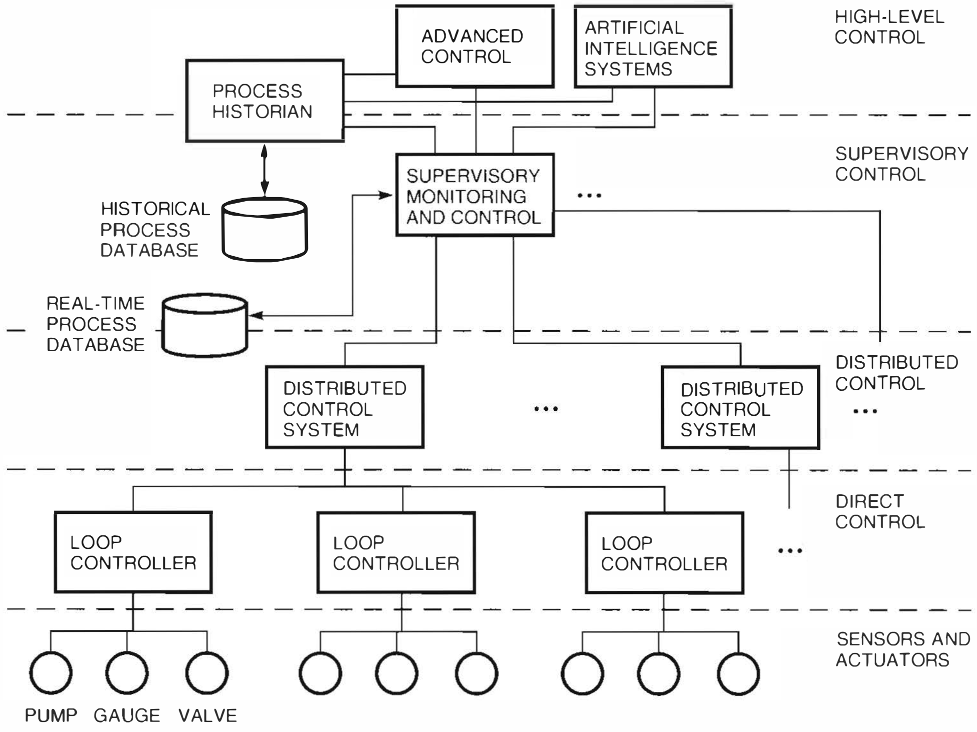


Figure 1 Typical Levels of Control in a Process Plant

problem or alarm conditions and modification of the process to change the product are effected through the console.

Process operators are responsible for maintaining the routine operation of a plant. Operators use the control system to change process parameters in order to produce different mixes or variants of the product, or to respond to an equipment failure by rerouting material around nonoperational process equipment.

To perform their functions, manufacturing plant production and engineering support personnel (e.g., control engineers, process engineers, production supervisors, production planners, maintenance supervisors, and manufacturing engineers) also need access to information in the control and supervisory systems. These professionals regularly access information contained in multiple manufacturing systems and have an occasional interest in particular measurements or parameters within other parts of the process. The functions of these manufacturing plant personnel include

- Complex problem analysis and solution. Locating sources of product or process variation involves analyzing information from different parts of the process that may be under the control of different automation systems. Comparing the flow that exits one part of the process with the flow that then enters the subsequent part, for example, could disclose a faulty flow meter, a previously unknown temperature control problem, or a leak.

- Product improvement. Improving product quality and consistency involves investigating how the product is affected by existing variations in the production process. For example, investigation may involve the study of a process variable that cannot be measured directly but can be calculated from the values of other process variables. Examining sets of variables over time and exploring possible relationships may result in discovering combinations of process variables that yield unexpected effects on product attributes.

- Process improvement. Improvements in process yield and process reliability and reduction of waste and hazardous by-products may involve the study of historical data values from the process. Studying measurements obtained from multiple control systems may also result in process improvements.

- Resource optimization. Usually, process plants are capable of producing different grades of product, as well as mixtures of end products. An oil refinery, for example, produces various grades of fuel oil and also home heating and lubricating oils, all from a single process. While the operators adjust the equipment to control the product mix, a process planner or production manager determines the best production schedule based on customer orders and the efficient use of the process equipment.

Process information is available to operators and engineers who are trained to work with the various control and management systems in the plant. Using proprietary tools for each system allows reports to be generated and specific types of analyses to be performed on the data contained within each of these systems. However, extracting the data from these systems to an engineer's desktop for analysis by generic tools, such as spreadsheets and statistical analysis packages, is difficult or even impossible. Lack of console- and tool-specific training is another obstacle to accessing process information.

Manufacturing Process Information Systems and Desktop Systems: Goals and Barriers

Production and engineering support personnel want to be able to use the desktop tools of their choice to explore and analyze data from manufacturing systems. Spreadsheets, simulation tools, report generators, visualization tools, statistical analysis tools, planning tools, charting tools, and graphic-generation tools have all become accepted parts of the array of computer-aided techniques and tools available to the contemporary knowledge worker. The interactive, easy-to-use graphical user interface, which can run on relatively inexpensive platforms under the complete control of the end user, has not only encouraged the wide use of these desktop tools but also enhanced their effectiveness. These tools stimulate professionals to creatively explore the character of large amounts of data and thus support the discovery of previously unexpected patterns and relationships.

The further an end user's primary function is from production, the more likely it is that such a user will want access to multiple systems. System interfaces, which may differ widely and are generally oriented toward production use, discourage users from making ad hoc inquiries into the system.

Consequently, manufacturing system data may not be easily accessible to users of the many desktop tools available for such purposes as decision support, research, analysis, and simulation.

Today, the use of data from the manufacturing process in planning, reporting, and managing the operation of a plant is hampered by the difficulty in accessing the data from plant control and process information systems. It is typical for a production supervisor who needs data from a control system to request the data from a process operator. Once in hand, the data is then manually entered into a spreadsheet or other desktop tool for analysis. The results of the analysis often require entering new parameter values into the control system. This task is typically performed by another person, trained to use the control system, who transcribes the values from a hard copy of the tool's output. The process is time-consuming, costly, and error prone. Problem-solving activities are limited to those that can justify the trouble and expense involved in simply accessing the data.

Existing Integration Efforts

The desire to use data from the control systems to analyze and improve the understanding and control of the manufacturing process has spawned a variety of efforts since the late 1980s. This work has attempted to ease the transfer of information between computing systems and control systems. However, the resulting products and standards are not oriented toward supporting ad hoc inquiries and, therefore, are not widely used.

Many currently available manufacturing systems may be connected to the plant network, but without standard higher-level interfaces, access to these systems remains limited.[3-7] Through such network connections, some manufacturing systems provide limited access to OpenVMS and/or DOS system users. However, the access is typically restricted to the use of unique, proprietary programming interfaces or to proprietary tools targeted at performing a manufacturing-related function, such as statistical quality control. Usually, interfaces are supplied only on a specific operating system or on limited versions of a specific operating system.

In some systems, it is possible to extract a table of data values into a file using a common representation and file format (such as Lotus Development Corporation's WK1) that can then be imported into a spreadsheet on an IBM-compatible PC. This technique obviates the need for hard-copy output and simplifies transcription but still requires that a specialist extract the data using proprietary interfaces. In addition, the data may need to be converted from string to numeric format to be usable within a particular spreadsheet.

The International Organization for Standardization standard *Manufacturing Messaging Specification* (IS9506 or MMS) addresses the problem of data exchange between applications and dedicated manufacturing systems (referred to in the standard as manufacturing devices).[8] Although some manufacturers of programmable controllers (that is, dedicated control systems that are primarily used in discrete manufacturing industries) offer MMS capabilities, the process industry manufacturers and their control system suppliers have not widely accepted MMS. Use of the standard has been perceived as expensive, inefficient, and oriented primarily toward the needs of discrete manufacturing. A committee of the Instrument Society of America (ISA) is developing a companion standard (ISA 72.02) to use with MMS in communicating with distributed control systems in process manufacturing.[9] An important aspect of this proposed standard is a data model that describes the organization and types of data in a distributed control system.

Requirements for Integration

Digital designed the DEC @aGlance architecture not to be a generic application integration mechanism but rather to support the integration of popular desktop tools with manufacturing process information systems. An application that complies with the architecture can be installed on any system within a network, run, and immediately exchange data with other compliant applications. Some key characteristics of the environment that helped to drive the architecture are

- Multiple vendors. Although, MS-DOS personal computers are the most popular desktop environment, VAXstation, Macintosh, and UNIX workstations have a clear presence in particular departments and in certain large customer sites.
- Multiple software developers. The applications to be integrated are products of many companies that build manufacturing systems and desktop tools. The software development groups in these companies focus on core application and human interface issues rather than on integration issues.

- A large variety of desktop applications and user interfaces. Each class of desktop application has a different way of interacting with users. Spreadsheets, for example, have very different user interfaces from statistical packages and data visualization packages. Some applications have elaborate macro languages, whereas others are almost entirely graphically driven.

- Multiple types of large networks. In the typical process manufacturing facility, large networks are already in place. While many plants use DECnet for their network, an increasing number of plants are choosing to use the transmission control protocol/internet protocol (TCP/IP), and some plan to migrate to Open Systems Interconnection (OSI) networks (including Digital's DECnet Phase V) from multiple vendors. PC LANs are also becoming popular.

- Conservative computing strategies. Large manufacturing facilities cannot afford to halt operation to make major changes in their production-related computing systems and networks. Such facilities look to standards-based products as a way of achieving stability and of ensuring confidence in the longevity of a particular technology.

Architectural Issues

Simply stated, the problem that the DEC @aGlance architecture attempts to address is, how can a set of existing applications running on heterogeneous platforms, distributed across a variety of networks, and developed by different vendors (with only peripheral interest in integration) be easily integrated? A good understanding of both the nature of the applications involved and how end users would use them if they were integrated is important for evaluating potential answers to the question.

The applications that we considered integrating can be divided into two groups: those that "own" manufacturing data, i.e., the manufacturing control systems, and those that are consumers of that data, i.e., the desktop tools. From the viewpoint of an end user, some aspects of the relationship between a desktop tool and a manufacturing control application must be considered in order to accomplish work goals. End users in this environment are primarily concerned about the manufacturing process, the equipment controlling the process, and the state of materials within the process. These users have little or no interest in such aspects as network topologies and protocols, operating systems, and byte ordering on different hardware platforms.

Some major concerns of the end user that the architecture should address are

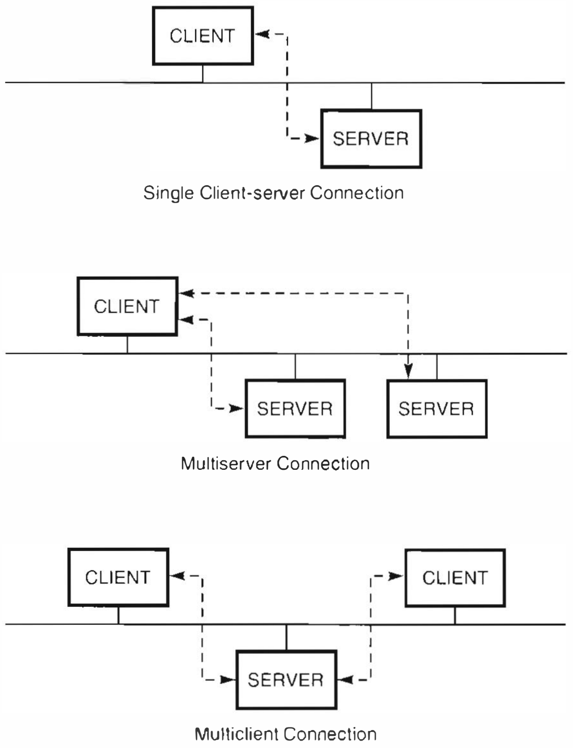
- The identity of the manufacturing control system. Generally, a large plant is controlled through the use of several control systems, each of which might control a part of the process, such as refining or packaging, or an aspect of the plant operation, such as steam distribution or waste reprocessing. A particular data point resides in a single manufacturing control system. The user should be able to specify precisely which manufacturing system is to supply the data values. The architecture should be capable of establishing a relationship with the specific application that owns the data of interest to the user. The end user should not have to specify either the network node, the operating system, or the hardware platform on which the application is running. Neither should the end user have to specify the network communication protocols required.

- The length of the relationship between the desktop tool and the manufacturing control application. The relationship should be able to remain active for multiple transactions to allow end users to work interactively with desktop tools to explore possibilities. For example, end users may want to examine different data points or the same data point over various time intervals. Thus, usage of a desktop tool could involve multiple requests for data from a manufacturing control application. Establishing a relationship between applications over a network is time-consuming, and therefore establishing long-lived relationships would be advantageous. The ability to continuously monitor a set of points and have their values reported on a time or change basis is another desirable feature that would require the establishment of long-lived relationships.

- Multiple access to the applications. Application relationships should not be exclusive. Each application should be able to have concurrent relationships with several partner applications. Each desktop tool may require data from several manufacturing systems, and conversely, several users of desktop tools may need to access the same control system simultaneously. The relationships between desktop tools and manufacturing control systems is illustrated in Figure 2.

- The data model. Applications should agree about how to reference data and about data types. Within the context of this environment, a relatively simplc data model exists in the draft standard ISA 72.02. Data should always be converted to types appropriate to the local system and to the application. A spreadsheet user should not have to manually convert strings into numeric values.

- The user interface. Application integration should not require the use of any particular desktop user interface, such as the X Window System or DECwindows software, or even the existence of a windowing system. Also, the user interface of the manufacturing data application should be of no concern to the desktop user.



Single Client-server Connection

Multiserver Connection

Multiclient Connection

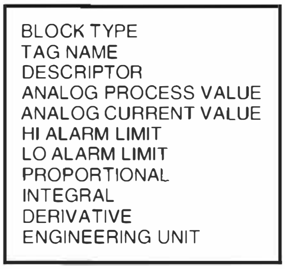
Figure 2 Relationships between Desktop Tools and Manufacturing Control Systems

Usage Model

To help us understand how a user might go about employing the capabilities that we were considering, we developed a simple usage model. We based the model on the scenario that an end user makes a series of ad hoc inquiries into the state of a process. We assumed that the user was familiar with the manufacturing process but not necessarily expert in all the details of the process. The user would know, for example, what the major areas of the plant were called and what functions they performed but might not know the internal reference identifier of every flow meter in each control system. We focused on how the user of a spreadsheet tool might reasonably expect to proceed to get data into a spreadsheet and how services that we might provide could aid in exploring the data.

The information within a manufacturing system consists of the many parameters and measurements that the system uses to monitor and control the process. Generally, this data is organized into blocks, each one related to a particular part of the process, such as flow, level, temperature, or pressure. As the typical data block in Figure 3 illustrates, every block has a unique name or tag that can be used for reference purposes.

In control systems, tag names are assigned as part of the configuration. Large plants use a naming convention to ensure the assignment of unique tag names to the thousands of blocks spread throughout the plant and over several control systems. In addition to the tag, the block contains attributes such as the parameters of the control algorithm, measured input values, unit conversion algorithm identifiers. The data model proposed by the ISA 72.02 committee describes seven types of blocks, each with a standard set of attributes with associated names and data types.



BLOCK TYPE
TAG NAME
DESCRIPTOR
ANALOG PROCESS VALUE
ANALOG CURRENT VALUE
HI ALARM LIMIT
LO ALARM LIMIT
PROPORTIONAL
INTEGRAL
DERIVATIVE
ENGINEERING UNIT

Figure 3 A Typical Data Block in a Manufacturing Control System

This usage model allows a user to easily determine the tag names recognized by a particular manufacturing system. To examine the data values associated with a specific tag, the user needs to know the valid attributes. (All blocks do not have the same attributes, e.g., an analog loop control block has more attributes than a simple digital monitoring block.) Once the tag names and their valid attributes are known, the user can inquire about current values as well as historical values.

The use of operating prototypes, including simulated servers and a simple spreadsheet, advanced the development of the usage model. The prototypes were shared with potential end users and application developers at customer visits and industry trade shows. Feedback obtained from demonstrations and discussions of the usage model helped expand and refine the services.

Architecture

The DEC @aGlance architecture defines two kinds of applications, a set of services for accessing data in the control systems, a data specification model, and some basic types of data. The application classes are (1) manufacturing data servers and (2) clients. Typical manufacturing data servers are the manufacturing control system applications. Typical clients include desktop tools such as spreadsheets and statistical analysis tools, as well as production planning, production scheduling, and other production management applications. An application may be a client in relation to one application and a server in relation to another.

A data point is specified to DEC @aGlance applications by the name of a server, a tag name, and an attribute name. A data point has a current value and may also have historical values (if the manufacturing system has a historian capability). A current value is the most recent available value of a parameter or measurement within the system. A historical value is a value that the data point had at some time in the past. A historical value is specified by the name of a server, a tag name, an attribute name, and the time associated with the value.

The services defined by the DEC @aGlance architecture fall into one of four functional categories: configuration information, data value exchange, monitoring, or management. Each service defines an operation that may be requested by one application of a partner application. The services defined are not necessarily the same functions that an end user requests.

Configuration Information

One service is defined for requesting the tag names that the server finds in the control system's database. An additional service returns a list of attribute names that are defined for a specified tag name or a list of tag names.

Data Value Exchange

Services are defined for reading and for writing current and historical data point values. For current values, services support reading or writing either a list or a table of data point values. A read or write list request specifies pairs of tag names and attributes. A read or write request for a table of data point values specifies a list of names and a list of attributes. The table of data points consists of all tag names paired with their corresponding attributes. Both the list and the table requests can be used to read or write a single data point, collapsing to either a list or a table of one data point.

By using the DEC @aGlance services to get lists of tag names, attribute names, and data point values, and the name of a server, an end user can generate a wide range of ad hoc queries without knowing much about the control system in advance. A common data point attribute is the descriptor, which characterizes the function of the data point, e.g., south tank level. Thus, it is a fairly straightforward task to use DEC @aGlance services to build a list of tag names and descriptors that provide a basis for further inquiries.

The services for historical data values are defined to deal with tables of historical values for a list of data points. Historical data service requests specify a list of tag name and attribute pairs and a time specification that is applied to all the data points. The time specification consists of a start time, a time interval, and the number of intervals for which values are to be returned.

Monitoring

Monitoring is useful for reading the values of a set of data points at intervals in time or when a significant change in value occurs for any of the data points. A graphical display program can run on a desktop system and make minimal use of the network and computing resources while maintaining an accurate representation of what is occurring in the manufacturing process. Monitoring could also be used to update a spreadsheet at regular time intervals or whenever a particular process variable changes.

No standard definitions exist for what constitutes a significant change in value. Definitions supported for various systems include (a) detection of change outside of a specified range or "dead band," (b) change by more than some percentage of the previously reported value, and (c) change by more than some percentage of a fixed value. Therefore, the service is defined to support monitoring and reporting of changes on a time basis or on some other basis that is specific to the data server application. Whenever the requested monitor condition is fulfilled, the data server application uses a monitor update service to send the new data point values to the original client application. Since the server initiates monitor update requests, the usual relationship between the client and the server is temporarily reversed.

Management

Connection management services are provided to establish a connection, to terminate a connection, and to test a connection.

Implementation Considerations

Using existing networking and application integration technologies to implement the DEC @aGlance architecture was important both in terms of reducing development efforts and improving compatibility with existing environments. Technology used in the implementation had to provide as many as possible of the capabilities described in the architecture while imposing minimal restrictions on the end-user operating and network environments and on the developers of the applications. In addition, it was desirable that the underlying technologies offer capabilities that could support future enhancements to the DEC @aGlance architecture.

The DEC @aGlance architecture allows an existing desktop tool to be integrated with existing manufacturing control systems, as shown in Figure 4. The architecture effectively combines the functional capabilities of the desktop tool for analysis, visualization, computation, etc., with the capabilities of the manufacturing control system for monitoring and controlling a manufacturing process. The individual applications were, of course, originally designed and written without any knowledge of each other's existence. Therefore, to facilitate integration efforts, implementation of DEC @aGlance software should localize and minimize required changes to the applications.

A network protocol such as DECnet, the transmission control protocol/internet protocol (TCP/IP), or one of the local area network (LAN) protocols could have provided the network services required for DEC @aGlance's interapplication communications. However, this approach lacks a mechanism for locating servers on the network, requires DEC @aGlance to support the multiple network protocols that exist in the manufacturing environment, requires DEC @aGlance to include data type conversion between application platforms, and necessitates the development of monitoring and management tools unique to DEC @aGlance. A better approach is to use an existing product that is available on an appropriate set of platforms, supports an appropriate set of networks, and already solves these problems.

A remote procedure call (RPC) mechanism appears to have many of the capabilities that the DEC @aGlance architecture requires. RPC

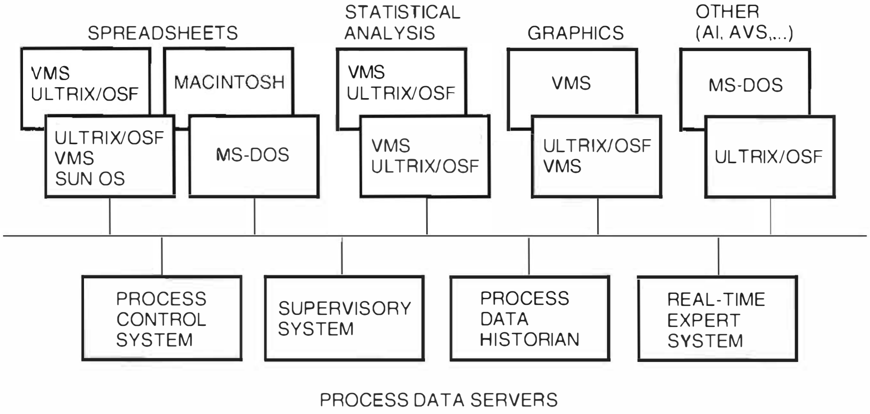


Figure 4 Integrating Desktop Tools and Manufacturing Systems

mechanisms provide for location of a partner or server application, and they provide data type conversion and reliable network services. The RPC model of application integration, however, is actually more appropriate for the distribution of a single application across multiple systems in a network. This use implies a simple, static relationship between the parts of an application: one part is always a client that requests the execution of a procedure, and the other part is always an RPC server that executes the procedure and returns the results. In such a relationship, each request generates a single response. This model would be poorly suited for supporting the DEC @aGlance monitoring service. When DEC @aGlance was being developed, no commercially available RPC implementation ran on the key platforms, the OpenVMS and Microsoft Windows environments. Furthermore, no one had announced their intention to produce a portable implementation that would be available on the wide range of platforms that we considered important for future versions of DEC @aGlance software.

Digital's ACA Services was chosen as the basis for implementing DEC @aGlance software because it implements an application integration model that closely matches the requirements of the DEC @aGlance environment. ACA Services supplies many capabilities required of the integration mechanism including

- Abstraction of functions from implementations
- The ability to encapsulate existing applications
- Location of partner applications on a variety of networks
- Establishment and management of reliable, long-lived communication links
- The ability to easily add new applications to the system
- The ability to easily install new versions of existing applications in the system
- The correct handling of data type conversions between heterogeneous systems
- Commercial availability of portable interfaces on OpenVMS, Microsoft Windows, Macintosh, and a wide variety of UNIX platforms from multiple vendors

The class hierarchy capabilities of ACA Services allow the creation of new combinations of applications integrated to provide new capabilities without additional coding. Thus, a new class of server can be defined to offer the capabilities of a DEC @aGlance data server as well as additional capabilities. The older DEC @aGlance servers would actually provide the DEC @aGlance services while, transparent to the client applications, the new server would make the new capabilities available.

ACA Services has been selected as a major component of the Object Management Group's (OMG) Object Request Broker, which in turn has been selected as a part of the Open Software Foundation's (OSF) Distributed Computing Environment (DCE). ACA Services is designed to be independent of the type of network that provides the interapplication communications services and currently works over both DECnet and TCP/IP networks, the networks most commonly found in manufacturing environments. Therefore, applications using ACA Services need not be concerned about network communications.

ACA Services is supported on the OpenVMS, Microsoft Windows, Macintosh, and SunOS operating systems, the most often used platforms in this application space. In fact, ACA Services is the only application integration mechanism currently available on all these platforms. Moreover, ACA Services supports the kind of asynchronous services required by DEC @aGlance.

Although it provides many important components of the required integration service, ACA Services does not completely solve the integration problem. ACA Services is a tool intended to be used to integrate applications; it does not define the data model nor does it define the set of services that applications are to provide. Application integrators are expected to define (1) the classes of applications that provide sets of services, (2) the services, and (3) the meaning and type of data to be exchanged by applications using the services.

DEC @aGlance Software: The Tool Kit and Add-ins

As shown in the DEC @aGlance component diagram in Figure 5, DEC @aGlance software uses ACA Services as a basic application integration facility. Above ACA Services, DEC @aGlance adds definitions of a class of manufacturing data server applications (servers), a set of definitions of the services provided by the servers, and definitions of the data reference model.

ACA Services provides a general capability to integrate sets of applications. DEC @aGlance software provides a set of routines that are specifically

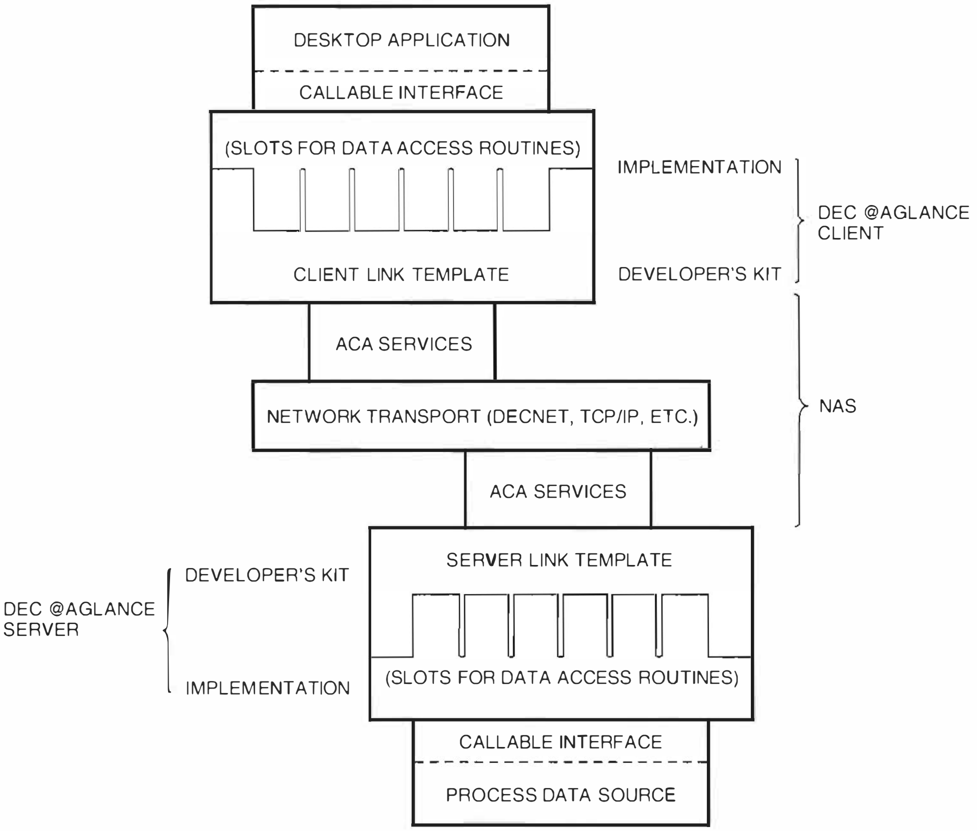


Figure 5 DEC @aGlance Components

designed to simplify the implementation of the set of services that DEC @aGlance supports. For server applications, DEC @aGlance software supplies a set of callback points, as well as callable routines for declaring callbacks, filtering strings, and supporting monitoring activities. For client applications, DEC @aGlance software supplies a set of callable routines for requesting each of the defined services, as well as callback points in support of monitor updates.

The DEC @aGlance server library also supports a test connectivity capability used to verify that an interapplication relationship can be established to the server application. This capability simplifies the diagnosis of problems encountered during both server development and client-server installation.

To reduce dependence upon properly written server code, the test connectivity capability operates entirely within the library. Thus, once a server calls the DEC @aGlance initialization routine, and if the server is still running, this service should function properly in response to requests from DEC @aGlance clients. Proper functioning includes verifying the installation and configuration of the network and of the ACA Services and DEC @aGlance run-time components of the systems on which the client and server applications reside.

Software add-ins, i.e., extensions, for two popular spreadsheet applications, Lotus 1-2-3 for Windows and Microsoft Excel for Windows, are also DEC @aGlance products. These add-ins allow users of the spreadsheets to request data from manufacturing data servers by means of the spreadsheets' macro facilities. The add-ins provide a dialog box to guide untrained users through the process of constructing a DEC @aGlance macro. Once built, a macro can be executed one or more times, modified if necessary, and saved in a worksheet for reuse at some other time.

Tool Kit

The tool kit was developed to encourage the rapid and successful development of DEC @aGlance applications by third parties. Successful applications are those that interoperate with other DEC @aGlance applications upon delivery to a customer site with no additional coding, no application recompilation, and no application rebuilding.

The key components of the tool kit are

- A DEC @aGlance client or server library
- Example code
- ACA Services definition files for the DEC @aGlance class and methods
- Simple test facilities
- The DEC @aGlance Programmer's Guide[10]

The ACA Services definition files contain the information required to define the manufacturing data server class and the services that members of the class support. Supplying the definitions in this form ensures strict consistency among all server and client developers with regard to these definitions. The routines in the DEC @aGlance client and server libraries use these definitions. The DEC @aGlance libraries contain all the code required to establish and maintain an ACA Services session.

Server Applications

A server application built with the tool kit has three major components: an initialization section, the control system-specific section, and the DEC @aGlance section. The initialization section simply declares the server's name to the DEC @aGlance application, declares a set of callback points, and enters a dispatch loop. The server name is the name that client applications can use to interact with this server. The callback points are the code entry points to which DEC @aGlance dispatches in response to the receipt of service requests from the client applications. For a server, callback points exist for the following services:

- Get a list of tag names
- Get a list of attribute names
- Get a list of data point values
- Get a table of data point values
- Put a list of data point values
- Put a table of data point values
- Get a table of historical values
- Put a list of historical values
- Register a monitor request
- Cancel a monitor request
- Initiate a session
- Terminate a session
- Execute a server-specific request
- Terminate the server

The control system-specific section consists of code modules that execute calls to the control system application programming interface (API). These modules have to convert parameters to and from the DEC @aGlance format and the control system-specific format. The entry point of each module is declared as a callback point during initialization.

In addition, callable routines are provided for sending monitor updates and for session management. The DEC @aGlance section of the server is contained entirely within a library of callable server routines. This section handles all interactions with ACA Services, including server registration and session management. It also handles the dispatch of incoming requests to the callback routines and a number of housekeeping tasks for which each server developer would otherwise have to develop and implement solutions. The DEC @aGlance section also responds to test connectivity requests.

Almost all vendors of manufacturing systems have applications that execute calls to the control system API, but such applications are typically driven off a command language or menu interface. Conversion of these applications to a DEC @aGlance server is relatively easy; some vendors have created a simple DEC @aGlance server in as little time as one day.

Client Applications

The typical DEC @aGlance client application is built on an existing desktop tool. Desktop tools provide a user interface for performing some class of generic function such as decision support, statistical analysis, quality control, or production scheduling. Other types of applications that could make use of process data, such as report generators, batch schedulers, and maintenance tracking systems, can also provide the basis of DEC @aGlance client applications. Adding DEC @aGlance support to an existing tool allows the user to treat data from DEC @aGlance manufacturing data servers like data entered manually or from other data sources.

A DEC @aGlance client application incorporates the DEC @aGlance client routine library, which provides callable routines for initialization and for each of the following DEC @aGlance services:

- Get a list of tag names
- Get a list of attribute names
- Get a list of data point values
- Get a table of data point values
- Put a list of data point values
- Put a table of data point values
- Get a table of historical values
- Put a list of historical values
- Initiate a monitor request
- Cancel a monitor request
- Initiate a session
- Terminate a session
- Execute a server-specific request
- Terminate the server
- Terminate the client

In addition, support routines help monitor updates.

To support the DEC @aGlance monitoring capability, a client application must have some server characteristics. Once a monitoring request has been initiated, the server issues monitor update requests when the monitoring condition is satisfied. The monitor update requests are received by the client application using the same callback mechanism that the server uses when servicing client requests.

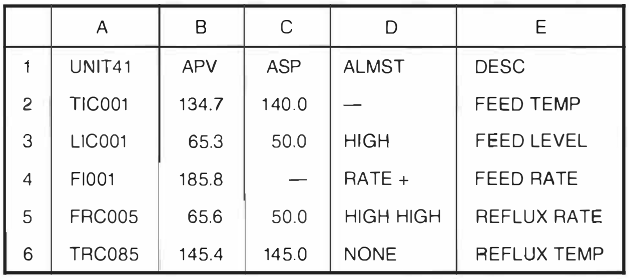
A typical client calls the DEC @aGlance initialization routine and then continues to perform its normal functions. When a DEC @aGlance service is requested through the user interface or other mechanism, the application simply formats the request and calls the appropriate DEC @aGlance service request routine. Upon completion of the routine, status (and if requested, data) is returned from the server application. If data is returned that is to be further processed by the client application, the application moves the data to its workspace in preparation for additional processing.

DEC @aGlance Lotus 1-2-3 for Windows and Microsoft Excel Add-ins

Whereas most manufacturing control systems provide a callable library that allows the development of applications that access the data in the system, some desktop tool applications have mechanisms that allow for extension of their capabilities in the field. Spreadsheet applications such as Lotus 1-2-3 and Microsoft Excel support the use of add-in modules to add external functions and external macro capabilities. Add-ins for these two spreadsheets are available as DEC @aGlance software products.

With the add-ins, spreadsheet users can access most DEC @aGlance services and thus can

- Fill a range of cells with a list of tag names from a server
- Fill a range of cells with a list of attribute names associated with a range of tag names in a server
- Fill a range of cells with a list of data point values
- Fill a range of cells with a table of data point values, as shown in Figure 6
- Write a list of data point values to a server
- Write a table of data point values to a server
- Fill a range of cells with a table of historical values for a specific time interval
- Write a list of historical values



	A	B	C	D	E
1	UNIT41	APV	ASP	ALMST	DESC
2	TIC001	134.7	140.0	—	FEED TEMP
3	LIC001	65.3	50.0	HIGH	FEED LEVEL
4	FI001	185.8	—	RATE +	FEED RATE
5	FRC005	65.6	50.0	HIGH HIGH	REFLUX RATE
6	TRC085	145.4	145.0	NONE	REFLUX TEMP

Figure 6 A Table of Data Point Values in a Spreadsheet

The interface for the add-ins was designed to support ad hoc inquiries. A dialog box guides the end user through the process of supplying the appropriate parameters for a selected function. Where appropriate, defaults are suggested based upon the previous inquiry.

Summary

DEC @aGlance software has been specifically designed to make it easy for users of desktop tools to access, explore, and analyze data from distributed control systems, supervisory control systems, and other common systems used to run manufacturing processes. An analysis of the information environment and the ways in which end users want to access the data led to the refinement of the architectural requirements. The analysis also led to the decision to use ACA Services as the appropriate mechanism for integrating desktop and manufacturing control applications. The creation of a usage model and rapid deployment of prototypes were instrumental in the analysis. To promote widespread availability of plug-compatible applications that use DEC @aGlance, a developer's tool kit was created. The tool kit contains libraries of DEC @aGlance routines that both simplify and encourage proper and consistent usage of ACA Services to integrate DEC @aGlance applications.

DEC @aGlance add-ins for the popular spreadsheet programs Lotus 1-2-3 for Windows and Microsoft Excel for Windows were developed also. With the add-in, users can interactively explore data in plant manufacturing control systems from within a familiar spreadsheet, as well as write reusable worksheet macros for performing repeated tasks like report generation.

Acknowledgments

The author gratefully acknowledges the contributions of the members of the DEC @aGlance development team: Judie Dow, Bob Harrison, Nick Miller, Ramesh Swaminathan, Patrick Taber, and the lead developer, Charlie Trageser. I would also like to thank Steve Dawson, for introducing our group to the problem and generally educating me about the process manufacturing environment; Chuck Kukla, for introducing me to his research on how people work in manufacturing and for his work with customers and control vendors that helped lead to the design of the product; Jim Thompson, for pushing and pulling all the strings that it took at every stage of the effort to bring the concept to a marketable product; and Mike Renzullo and Alan Ewald of the ACA Services Development Group, for their support.

References

1. This quotation was taken from the transcript of an interview conducted by C. Kukla et al., who have published the results of their study in "Usability Turning Technology into Tools," *Designing Effective Systems: A Tool Approach,* P. Adler and T. Winograd, eds. (New York, NY: Oxford University Press, 1992).

2. *DEC ACA Services System Integrator and Programmer's Guide* (Maynard, MA: Digital Equipment Corporation, Order No. AA-PFYUA-TE, 1992).

3. *CM50N User Manual,* Order No. CM11-320 (Phoenix, AZ: Honeywell Industrial Controls and Automation, 1991).

4. *Computer/Highway Interface Package (CHIP) User Guide,* Part No. D001093X012 (Marshalltown, IA: Fisher Controls International, Inc., 1987).

5. *AIM Connectivity Software User's Manual* (Houston, TX: W. R. Biles and Associates, Inc., 1992).

6. *S/2 SCADA System Description,* Document No. SD2.0001 (Dallas, TX: Texas Instruments, Industrial Systems Division, 1988).

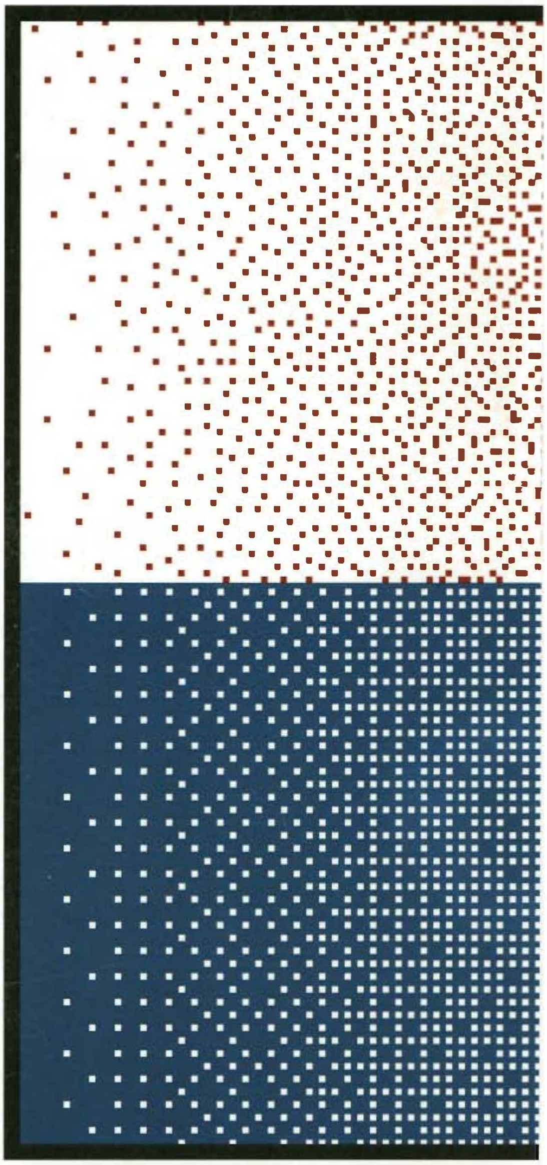
7. *PI System Plant Information System Technical Overview* (San Leandro, CA: Oil Systems, Inc., 1990).

8. *Manufacturing Messaging Specification,* ISO/IEC 9506 (Geneva: International Organization for Standardization/International Electrochemical Commission, 1990).

9. *Manufacturing Messaging Specification: Companion Standard for Process Control,* ISA 72.02 (Research Triangle Park, NC: Instrument Society of America, 1993).

10. *DEC @aGlance Programmer's Guide* (Maynard, MA: Digital Equipment Corporation, Order No. AA-PQB8A-TK, 1992).

digital™



ISSN 0898-901X