LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-637

A PROGRAMMING SYSTEM FOR THE DYNAMIC MANIPULATION OF TEMPORALLY SENSITIVE DATA

Christopher J. Lindblad

August, 1994

545 TECHNOLOGY SQUARE; CAMBRIDGE, MASSACHUSETTS 02139 (617) 253-5851

A Programming System for the Dynamic Manipulation of Temporally Sensitive Data

Christopher J. Lindblad

Telemedia Networks and Systems Group
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

In computer-participative multimedia applications, the computer not only manipulates media, but also digests it and performs independent actions based on media content. In this report I discuss an approach to the design of environments to support the development of computer-participative multimedia applications and I describe the implementation of the VuSystem, a computer-participative multimedia system built using this approach. The system is unique in that it combines the programming techniques of visualization systems and the temporal sensitivity of multimedia systems. I report measurements made of the performance of the VuSystem, which demonstrate its practicality. I conclude with a brief summary of users' experiences with the VuSystem, and suggests future directions for research in this area.

This research was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the United States Air Force (AFSC, Rome Laboratory) under contract No. F30602-92-C-0019, and by grants from Nynex and Digital Equipment Corporation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This report describes the *VuSystem*, a programming system for the software-based processing of temporally sensitive data. It runs on high performance computer systems not specifically designed for the manipulation of digital media, such as audio and video. The system is unique in that it combines the programming techniques of visualization systems and the temporal sensitivity of multimedia systems.

VuSystem applications have two components: one which does traditional *out-of-band* processing and one which does *in-band* processing. Out-of-band processing is the processing that performs the event-driven functions of a program. In-band processing is the processing performed on every video frame and audio fragment. In-band code is more elaborate in the VuSystem than in traditional multimedia systems [12, 17] because VuSystem applications perform sophisticated analysis of their input media data.

In the VuSystem, the in-band processing component is arranged into processing *modules* that pass dynamically-typed data *payloads* through input and output *ports*. The out-of-band component of the VuSystem is programmed in the Tool Command Language, or Tcl [26], an interpreted scripting language. Application code written in Tcl is responsible for creating and controlling the network of in-band media-processing modules, and controlling the graphical user-interface of the application.

The VuSystem is implemented on Unix workstations as a program that interprets an extended version of Tcl. In-band modules are implemented as C++ classes and are linked into this Tcl shell. Simple applications that use the default set of in-band modules are written as Tcl scripts. More complicated applications leverage customized modules that are linked into the shell.

VuSystem programs have a *media-flow* architecture: code that directly processes temporally sensitive data is divided into processing *modules* arranged in data processing *pipelines*. This architecture is similar to that of some visualization systems [29, 31], but is unique in that all data is held in dynamically-typed time-stamped *payloads*, and programs can be reconfigured while they run. Timestamps allow for media synchronization, and dynamic typing and reconfiguration allows programs to change their behavior based on the data being fed into them.

The VuSystem's design makes it particularly well suited as an application toolkit for distributed multimedia systems. In particular, the VuSystem is used as the application environment for the ViewStation hardware platform, a set of computers and programmable digital video processing devices connected together by a personal local-area network [4].

Because the VuSystem provides a rich environment of media processing modules linked together with a high-level scripting language, it is a particularly good foundation for the development of applications that require intelligent processing of media data. These applications are best called *computer-participative* multimedia applications, because the computer directly participates in the interpretation of the media data.

In rest of this introduction, I motivate the design of the VuSystem by describing some computer-participative multimedia applications that have been written in it. I also describe the ViewStation, the hardware and software platform for which the VuSystem is designed. I conclude the chapter with a statement of claims, and provide a roadmap of this report.

1.1 Computer-Participative Multimedia

The term *multimedia* generally refers to the capture, storage, retrieval and presentation of audio and video data using computers. Typical multimedia applications include online encyclopedias and video-conferencing systems. These applications are perhaps better classified as *computer-mediated* multimedia applications. The computer acts as a mediator between the application author and user in the case of the online encyclopedia, or between two users in the case of the video-conferencing application.

In contrast, *computer-participative* multimedia applications also perform analysis on their audio and video data input, and take actions based upon the analysis. For example, a program that watches television news shows and maintains an online database of stories organized by subject is a computer-participative multimedia application. The program must analyze the content of its input to sort it into stories. The computer is an *active participant* in the processing of audio and video data.

Current programming systems are inadequate for computer-participative multimedia applications. Commercial multimedia systems generally support the efficient storage, retrieval, and presentation, of pre-recorded video clips, but do not adequately support the direct processing of live media. Some experimental multimedia systems for Unix workstations have been developed, but provide only a limited range of operations on media data. Visualization systems allow a wide variety of operations on images, but they do not support temporally sensitive data.

1.2 Some Computer-Participative Applications

Several examples of computer-participative multimedia applications have been built with the VuSystem. *The Room Monitor* and *The Whiteboard Recorder* applications, for example, directly process live video. *The News Browser*, *The Joke Browser*, and *The Sports Browser* applications perform retrieval of pre-recorded multimedia based on automatically extracted content. Each application requires some sort of direct access to temporally sensitive data.

1.2.1 Applications That Process Live Video

Use of the VuSystem has revealed it to be a good platform for the investigation of concrete ways that computers may become more responsive to their human users. One user, William Stasior, is developing a prototype "Computerized Office Multimedia Assistant" (COMMA) with the VuSystem. COMMA assists its user by performing various tasks that require the analysis of live video [8].

The Room Monitor

Stasior has written *The Room Monitor*, which processes continuous video from a stationary camera in a room. It processes the live video to determine if the room is occupied or empty, and records video only when activity is detected above some threshold. It produces a series of video clips that summarize the activity in the room. A video browser (Figure 1.1) is used to view the segments. The video clips allow the user to check who was in the room and when.

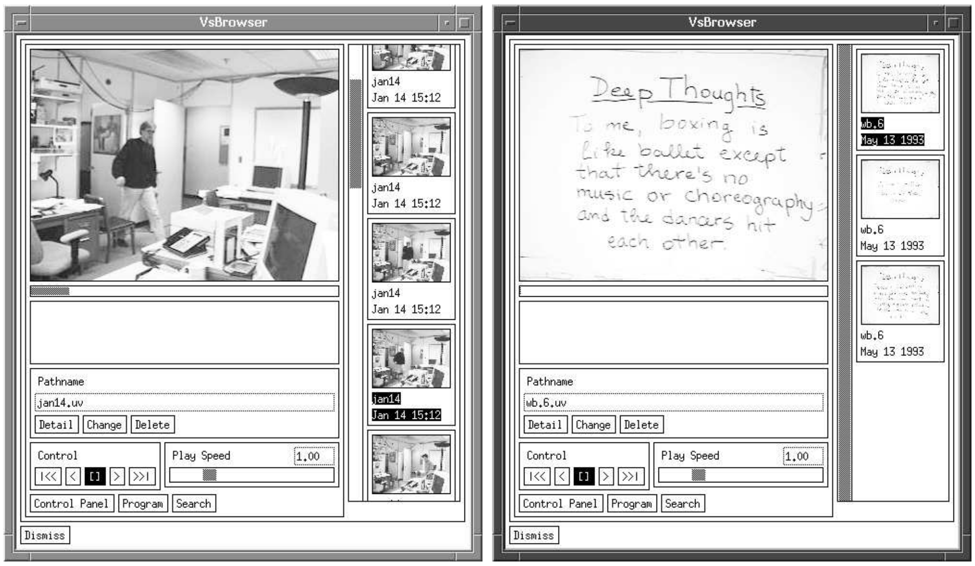


Figure 1.1: Browsers operating on the output of *The Room Monitor* and *The Whiteboard Recorder*, two applications that directly process live video.

The Whiteboard Recorder

Stasior has also written *The Whiteboard Recorder*, an application that keeps a history of changes to an office whiteboard. It works by taking continuous video from a stationary camera aimed at the whiteboard and filtering it. By following a simple set of rules, the filtering distills the video into a minimum set of images. A video browser can be used to view the saved images.

The whiteboard recorder uses motion analysis to distinguish between the person writing on the board and the writing itself. Live video captured from a fixed camera is processed so that transient image features are filtered out, and only relatively stationary features are retained. It distinguishes changes to the whiteboard due to writing, from changes due to erasing. The system saves away images that represent "peaks" in the information written on the board.

1.2.2 Content-Based Processing Of Television Programs

The VuSystem has been used to explore the potential of media processing applications to support content-based retrieval of pre-recorded television broadcasts. VuSystem users have developed content-based media browsers that use textual annotations that represent recognizable events in the video stream. These annotations are analyzed and processed to create higher level representations that may be meaningful to a human user. Finally, these representations are matched against user queries to generate an interactive presentation in the form of a browsable set of relevant video clips.

Annotations are generated through the recognition of audio or video cues from the media stream, or by the extraction of ancilliary information included in the stream, such as closed captions. *The Sports Browser*, *The News Browser*, and *The Joke Browser* are built on the processing of these annotations.

Figure 1.2: *The Sports Highlight Browser*, an application that supports content-based retrieval of video segments from a television program using video analysis.

The Sports Highlight Browser

Stasior has developed *The Sports Highlight Browser*, which segments a recorded sporting news telecast into a set of video clips, each of which represents highlights of a particular sporting event. Video highlights of a particular game can be requested with a browser (Figure 1.2).

The annotation analyzer is built with assumptions about the format of a sports telecast. In particular, this analyzer depends on the news cliché of first an anchor person, then a set of narrated video clips, and finally a scoreboard graphic. The analyzer groups into a highlight the video sequence that falls between two scoreboard graphics. The analyzer labels each highlight with the names of the teams that competed.

The Broadcast Television News Browser

The Broadcast Television News Browser provides interactive access to a simple database of news articles. Live news programs such as CNN Headline News are automatically captured to disk at regular intervals. The stories are viewed with a video browser program (Figure 1.3).

News stories that are closed-captioned can be retrieved based on their content. Many broadcast television programs are closed-captioned for the hearing-imparied. Closed-captions provide a text translation of the audio component of the program — a significant amount of information. Closed-caption capturing code runs in the Vidboard [9]. The caption information is extracted from the digitized video signal and converted into a common format so that modules capable of processing it can be constructed.

The news browser makes direct use of the closed-captioned annotations. A text search specification supplied by the user causes the browser to jump to stories with captions that match.

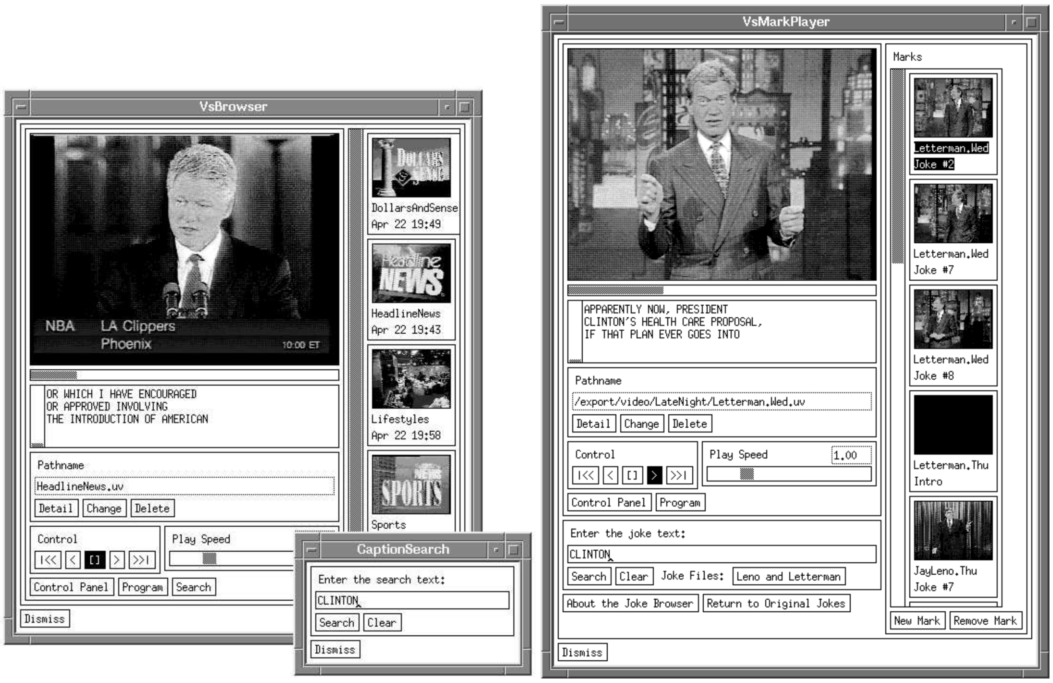


Figure 1.3: *The Broadcast Television News Browser* and *The Joke Browser*, two applications that support content-based retrieval of video segments from closed-captioned television programs.

The Joke Browser

David Bacher has developed *The Joke Browser*, which further demonstrates the potential of content-based media processing using closed-captions [3]. It records late-night talk show monologues, and segments them into jokes by processing the closed-captioned text. A special browser program (Figure 1.3), is queried to select all the jokes on a certain topic that have been made in the last week.

The Joke Browser extracts information from a recorded monologue through the analysis of the closed-caption data. In addition to the text of the jokes, the closed-captions contain hints to the presence of audience laughter and applause. A joke parsing module groups captions into jokes. This module is program specific, as it uses knowledge of the format of a particular program to make its grouping decisions.

1.3 The ViewStation

The ViewStation project [4] integrates the technologies of broadband networking and distributed computing with those of digital video to produce systems for video-intensive computing. The ViewStation platform is composed of a set of computers and programmable digital video processing devices connected together in a personal local-area network.

The purpose of the project is to build a local-area distributed multimedia system. It takes a different approach towards multimedia devices, where these devices are treated as general purpose peripherals, providing a uniform hardware interface to the network. Thus the data generated from all such devices are treated by hosts in a homogeneous manner.

The project focuses on getting real-time data such as voice and video from the network

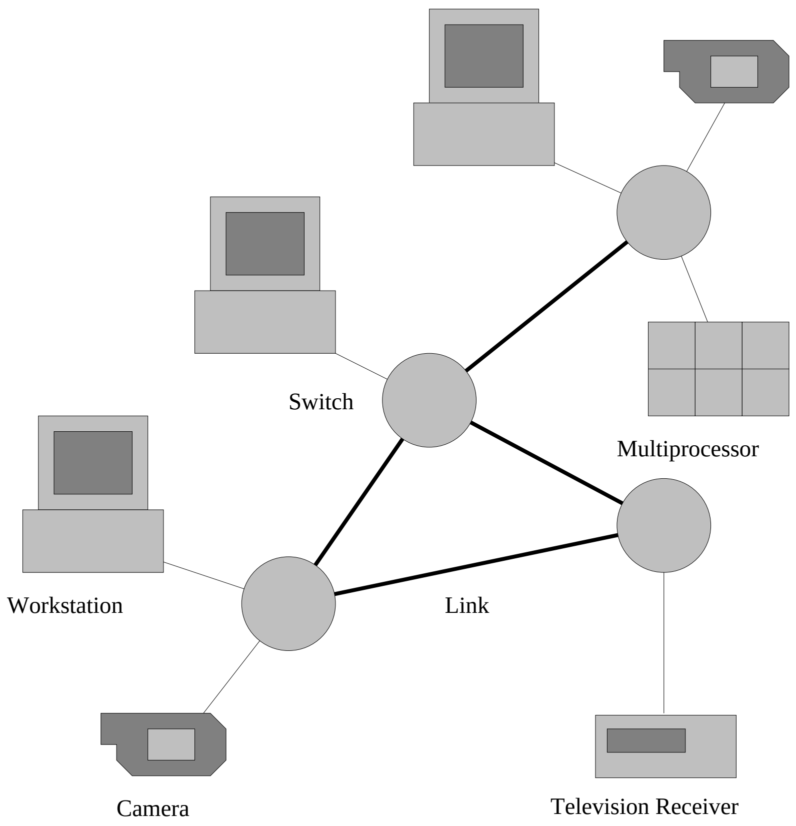


Figure 1.4: The ViewStation Architecture.

all the way to the application. Since the ViewStation takes a software-intensive approach to multimedia, the VuNet and custom multimedia hardware were designed to provide efficient support for software-driven handling of multimedia streams.

ViewStation applications are built with the VuSystem. The VuSystem provides simple scheduling and resource management functions to allow intelligent media-processing applications to run on workstations not specifically designed for multimedia. The VuSystem is uniquely suited for rapid development of applications that perform intelligent processing of live media on the ViewStation. It runs on Unix workstations connected to the VuNet, and is used to build applications that use VuNet peripherals. In particular, VuSystem applications make use of the Vidboard [9], a video capture device that resides on the VuNet.

1.4 Statement Of Claims

I have identified a class of multimedia applications in which the computer performs tasks requiring the direct processing of multimedia data, as well as the capture, storage, retrieval, and display tasks of traditional multimedia applications. Members of the class are best called *computer-participative* multimedia applications, because in them the computer directly participates in the interpretation of the multimedia data. These applications require more support than is provided by traditional multimedia toolkits.

Applications of this type have been implemented in the past. Bender and Chenais have developed a system that digests television news broadcasts and annotates them with newspaper articles [33]. Abramson and Bender have developed a system in which self-aware multimedia objects alter their content to fit user preferences [15]. In this report I build on this work by establishing a framework for investigating computer-participative multimedia applications as a class.

To support the development of computer-participative multimedia applications, I designed and built the VuSystem, a prototype software development environment for the development of applications that directly manipulate temporally sensitive data. The system provides simple scheduling and resource management functions to allow intelligent media-processing applications to run on ordinary Unix workstations. It is uniquely suited for rapid development of applications that perform intelligent processing of live media.

In the VuSystem, application code is split into two partitions: an *out-of-band* partition to handle user interfaces and other event-driven program functions; and an *in-band* partition to perform the periodic media processing operations. The architecture for each partition was designed separately: the in-band partition for high performance, and the out-of-band partition for ease of programming. This arrangement enables the rapid development of complex and intelligent media-processing applications.

Code in the in-band partition of the VuSystem is arranged into processing *modules* that logically pass dynamically-typed data *payloads* though input and output *ports*. This structure provides for the modularity and extensibility necessary for the support of computer-participative multimedia applications, but still retains a high degree of efficiency and temporal sensitivity.

The out-of-band partition of the VuSystem has more relaxed performance constraints, since it is designed to handle relatively infrequent events. An interpreted scripting language, such as the Tool Command Language, or Tcl [26], is an appropriate programming language for the out-of-band partition. Out-of-band VuSystem application code written in Tcl is responsible for creating and controlling the network of in-band media-processing modules, as well as controlling the graphical user-interface. Tcl provides a high-level language interface to the VuSystem that encourages the rapid development of prototype computer-participative multimedia applications.

Following the construction of the VuSystem prototype, the suitability of the system was verified through the development of applications that use the environment. The applications are based on an extensive library of primitive modules that I and others have built. The *Puzzle* program, which displays a video picture scrambled into a 16-square puzzle, is used as an example throughout this report. Other applications already mentioned in this introduction include *The Room Monitor*, *The Whiteboard Recorder*, *The News Browser*, *The Joke Browser*, and *The Sports Browser*. These applications demonstrate the benefits of computer-participative multimedia.

Measured performance of VuSystem applications reveal my approach to be practical. The VuSystem run-time component operates with low overhead. Representative VuSystem in-band processing modules are efficient. The VuSystem scheduler can cause operations to occur at reasonably precise times. The system has enough thoughput to support full-motion video. Measurements verify that the VuSystem meets perceptual-time constraints sufficiently to support media-processing applications.

1.5 This Report

This report is divided into eight chapters. Chapter 1 is this introduction. Chapter 2 contains a summary of relevant previous work. In Chapter 3, I describe my design approach. Chapter 4 describes the programming of VuSystem applications using an interpreted scripting language. Chapter 5 describes the programming of VuSystem media processing modules. Chapter 6 describes how the scripting language and the modules interact. Chapter 7 shows how the VuSystem performs, and Chapter 8 presents the conclusions.

The appendicies of this report provide reference documentation for the VuSystem. Appendix A documents the predefined modules in the VuSystem. Appendix B documents application support in the VuSystem, Appendix C documents module programming support, and Appendix D documents graphical-user-interface programming in the VuSystem. Appendix E shows sample application and module code for the *Puzzle* program, a simple VuSystem application.

Chapter 2

Perspective

In this chapter I discuss previous work that is relevant to the design of systems for the intelligent processing of live media data on ordinary computer workstations. Work in this area is best divided into three areas: commercial multimedia systems, research multimedia systems, and visualization systems. Work done in each of these areas is applicable to the problem.

2.1 Commercial Multimedia Systems

Commercial multimedia systems exist primarily to support the timely storage, retrieval, and presentation, of pre-recorded video clips. These systems are optimized for the efficient display of pre-recorded video sequences. They do not adequately support the direct processing of live video.

2.1.1 Apple Quicktime

Apple Computer has developed *Quicktime*, a tool kit for the manipulation of time-based data within the Macintosh environment [12]. It is the most popular system for multimedia applications today.

In Quicktime, time-based data is referred to as *movies*. Applications allow users to display, edit, copy and paste movies and movie data. These applications manipulate Quicktime *components* (Table 2.1) through *managers* (Table 2.2). Quicktime components support defined sets of features and present specified functional interfaces to their client applications, while Quicktime managers provide access to system facilities. Quicktime defines time-coordinate systems that anchor movies and their media to a common

Component Type	Function
Clock	Provides timing signals for Quicktime applications.
Image compressor	Compresses and de-compresses image data.
Movie controller	Allows applications to play movies using a standard user interface.
Sequence grabber	Allows applications to obtain video and sound.
Sequence grabber channel	Provides interfaces between a sequence grabber and the external data source and writes the data into quicktime movies.
Video digitizer	Allows applications to control an external device which produces video.

Table 2.1: The predefined Quicktime component types.

Manager	Function
Movie Toolbox	Supports retrieval and manipulation of time-based media stored in movies.
Image Compression Manager	Provides an interface abstraction to compression and decompression resources.
Component Manager	Allows the developer to define and register code resources and communicate with them using a standard interface.

Table 2.2: The Quicktime managers.

timescale, the number of time units per second.

Apple Quicktime deals with the scheduling and resource allocation issues of audio and video delivery within the constraints of the Macintosh environment. Its component architecture provides the extensibility necessary for direct media processing programs, but it is poorly documented and only works within the Macintosh environment.

2.1.2 Microsoft Video For Windows

Microsoft *Video For Windows* enables users of Microsoft Windows to capture, edit and play back video sequences without specialized hardware [17]. It provides efficient playback of video sequences from hard disk or CD-ROM, regardless of the capabilities of the PC.

Video For Windows is based on the multimedia features of Windows 3.1. Media data is stored in the resource interchange *audio video interleaved* file format, or AVI. The AVI file player supports Microsoft's media control interface for digital video, DV-MCI.

Like Quicktime, Video For Windows is designed to be *scalable*. During video playback, it automatically takes advantage of all the capabilities of the system upon which it is running, resulting in digital video with color and motion that improve with the performance level of the machine. On slower computers, video playback degrades gracefully, displaying as much information as possible and always maintaining audio continuity.

For the multimedia author, Video For Windows provides a set of tools for creating software-only digital video sequences. The tools can be used by professional multimedia title developers as well as individual users.

For the software developer, Video for Windows provides interfaces for the creation of tools for capturing, editing, enhancing, and utilizing video sequences. The Video for Windows architecture also provides hooks for adding video capture drivers, and video codecs. The codecs can be software-only, hardware assisted, or hardware only.

Video For Windows is designed for the efficient capture, edit and playback of audio and video data within the Windows environment on personal computers. It is not designed as an architecture for the direct processing of live media. It is somewhat extensible, in that video capture drivers and codecs can be added, but it does not effectively support applications that perform extensive processing on their media input, nor does it modify their behavior based on their input.

2.2 Research Multimedia Systems

Networked multitasking workstations running Unix have been found to be useful because applications developed for them can run on a variety of platforms with a variety of capabilities. It would be good if audio and video applications written for Unix could retain the portability and scalability that has made Unix so useful. However, since audio and video data are temporally sensitive, audio and video applications need guarantees on the computational resources available so that the capture or playback of the data is smooth and timely. These guarantees cannot be met by traditional Unix services.

2.2.1 Abstractions for Continuous Media

Anderson et al describe a set of *Abstractions for Continuous Media* (ACME) in [24]. The ACME system provides shared network-transparent access to real-time audio and video hardware. The system provides mechanisms for resource management and scheduling for audio and video applications. In the ACME system, each workstation runs an ACME server, in which all the processing of real-time data is performed.

The ACME system is not distributed, but instead works as one server on a workstation. Applications exist as out-of-band clients that make configuration requests of the server, which performs all the in-band data processing. These applications communicate to the server through a control protocol.

In ACME, since media data never leaves the server, there is no possibility of applications being able to directly process media data. ACME provides no in-band processing extensibility. No new in-band data manipulations can be implemented by the application programmer. Any new in-band functionality can only be introduced through modification of an ACME server.

2.2.2 Comet

Anderson et al address some of the limitations of ACME with *Comet* [19]. Comet provides a high-level distributed application programming interface to ACME. In Comet, applications exist as processing nodes interconnected by data streams. Comet maps the abstract description of a processing network into a network with physical components. In Comet, the processing components are implemented by ACME servers, and data communication is performed by TCP streams.

Comet provides a distributed processing capability that ACME alone cannot provide, but still suffers from the lack of in-band extensiblity. All the in-band functionality of Comet is provided by ACME servers. New applications can provide new out-of-band functionality, but no new in-band functionality. New in-band functionality can only be introduced through an ACME server.

2.3 Visualization Systems

Visualization systems allow a wide variety of operations on sequences of images. They provide a library of image processing modules that can be hooked together to transform a sequence of source images stored in individual files to a sequence of result images. They also provide a graphical programming system that can be used to combine processing modules into programs.

2.3.1 The Animation Production Environment

The *Animation Production Environment* (apE) is a flexible integrated graphics environment for the production and manipulation of computer graphic images [34]. Built for the Ohio Supercomputer Project, apE is designed for the manipulation of large scientific data sets through visualization. The apE programming model is similar to Comet's — applications are constructed through the assembling of processing modules with data and control connections.

The Animation Production Environment is extensible because new processing modules can be defined by the programmer, but apE does not handle the temporal manipulations of the data. The apE system operates without the anchoring of any component to real time. Since it has no basis in real time, apE does not provide for any resource scheduling based on real time.

2.3.2 Khoros

Developed at the University of New Mexico, *Khoros* is an integrated software development environment for information processing and data visualization [29]. It includes extensive data display and processing libraries, and a visual programming language. It runs on Unix workstations.

Khoros includes extensive application specific data display and processing libraries, providing support for image processing, digital signal processing, numerical analysis, data and file conversion, graphics display, and image display.

Cantata, the visual programming language of Khoros, is a graphically expressed, data-flow oriented language. With Cantata, scientists and engineers can assemble library modules into visualization programs without writing any code.

Khoros has no support for temporally sensitive data. Khoros programs cannot easily reconfigure themselves dynamically based upon their input. It is designed primarity for data processing and visualization, not for media-based intelligence.

2.3.3 The Application Visualization System

The *Application Visualization System* (AVS) is a suite of tools for the visualization and analysis of large computer-generated data sets [31]. AVS includes support for 2D plots and graphs, image processing, and interactive 3D rendering and volume visualization. It works on many major Unix workstations, supporting the full range of graphics hardware available on these platforms.

AVS includes a collection of hundreds of modules, and a developer can include custom code or link AVS to external applications. A module generator provides the developer with an environment for rapidly creating and maintaining AVS module code in an object oriented fashion. AVS also includes a visual programming environment where modules can be graphically connected together to build a visualization network, which becomes a customized application.

AVS includes an animation tool to assist in the generation of animated data visualizations. It can be used to generate image sequences from simulation output, to be displayed on a computer screen. This tool helps scientists and engineers visualize data through animation.

AVS is not designed for the direct processing of live media. It only supports the manipulation of visual data. It has no support for audio or other media. It has no notion of temporal sensitivity of its input. It is designed primarily to make computer data more understandable to scientists and engineers through visualization.

2.4 The Challenge

Commercial multimedia systems generally support the efficient storage, retrieval, and presentation, of pre-recorded video clips, but do not adequately support the direct processing of live media. Some experimental multimedia systems for Unix workstations have been developed, but provide only a limited range of operations on media data. Visualization systems allow a wide variety of operations on images, but they do not support temporally sensitive data.

The challenge is build a system that can support computer-participative multimedia applications. Like traditional multimedia systems, such a system should support temporally sensitive data. Additionally, it should provide an extensive set of operators that can be used to perform intelligent processing on the data. It also should be extensible, in that it should allow developers that use the system to define new operators. Finally, it should support elaborate control structures that intelligent multimedia applications would require.

System	In-band Extensibility	Temporal Sensitivity
Quicktime	bad	good
Video For Windows	bad	good
ACME	bad	good
apE	good	bad
Khoros	good	bad
AVS	good	bad

Table 2.3: Traditional multimedia systems provide good temporal sensitivity, but bad in-band extensibility, while visualization systems provide good in-band extensibility but bad temporal sensitivity.

In the next chapter I present the design approach I used for the VuSystem in response to this problem. I also describe the implementation of the VuSystem, and demonstrate that it is well suited to the development of computer-participative multimedia applications.

Chapter 3

Approach

In the VuSystem, application code is split into two partitions: an *out-of-band* partition to handle user interfaces and other event-driven program functions; and an *in-band* partition to perform the low-level media processing operations. The architecture for each partition was designed separately: the in-band partition for high performance, and the out-of-band partition for ease of programming. In this chapter, I describe this approach.

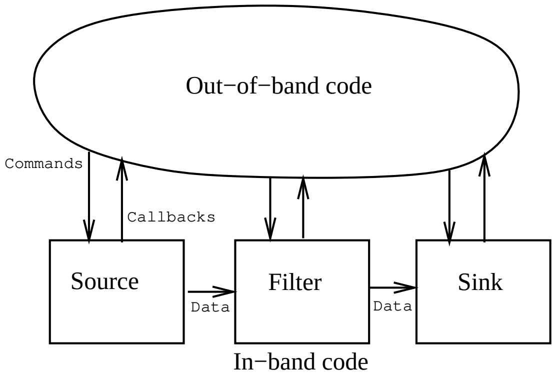


Figure 3.1: The structure of VuSystem applications.

VuSystem programs have what can be called a *media-flow* architecture: code that directly processes temporally sensitive data is divided into processing *modules* arranged in data processing *pipelines*. This architecture is similar to that of some visualization systems [29, 31], but is unique in that all data is held in dynamically-typed time-stamped *payloads*, and programs can be reconfigured while they run. Timestamps allow for media synchronization, and dynamic typing and reconfiguration allows programs to change their behavior based on the data being fed into them.

3.1 Multimedia Application Structure

The code in multimedia applications can be split into two classes: that which does traditional out-of-band processing and that which does in-band processing.

Out-of-band processing is that processing which performs the event-driven functions of a program. Code that performs the out-of-band processing in an application is the familiar event driven code typical of all interactive applications. This code awaits user input and other events and performs actions based on the events. Much of the out-of-band code rarely runs in a given session, because it is built to handle many possible contingencies. Software that performs out-of-band processing need not be extremely efficient, but should be easy to develop.

In-band processing is the processing performed on every video frame and audio fragment. It is any processing performed continuously on a running audio or video sequence. This code is characterized by repeated actions that occur many times a second. The in-band code usually is a small part of the overall application code, but consumes most of the time of an application run. Software that performs in-band processing should be very efficient, since it is running most of the time.

In-band code is more elaborate in computer-participative multimedia applications than in computer-mediated multimedia applications. A computer-mediated data capture application might simply move data from camera to disk. A computer-participative data capture application might analyze the data from the camera, and only save it on disk if a person were in the picture. The in-band code in the computer-participative application would be more elaborate, since it would need to analyze the video data to determine if a person were in the picture.

Toolkits for computer-participative multimedia applications support more *modularity* and *extensibility* in in-band processing than toolkits for computer-mediated multimedia applications. It is impossible for the toolkit developer to predict all the possible in-band operations that a computer-participative multimedia application might need, so the application developer must be supplied with a mechanism to extend any in-band processing capability.

In-band processing and out-of-band processing are best handled in separate partitions instead of together, because then choices of language and architecture can be made for each partition separately. For example, both in-band and out-of-band processing can be handled together in a single monolithic program, but the result would be suboptimal. If the program were entirely written in a C-like low-level language designed for efficiency, then the out-of-band processing would be too clumsy for rapid prototyping. If the program were completely written in a high-level Lisp-like language designed for rapid prototying, then the in-band processing would be inefficient.

3.2 Architecture Of The In-Band Partition

Since the in-band processing partition consumes more system resources and has tighter timing constraints than the out-of-band partition, care was taken with its design. The in-band architecture of the VuSystem is highly structured. Its design emphasizes efficiency, modularity, and extensibility. In the VuSystem, the in-band processing partition is arranged into processing *modules* which logically pass dynamically-typed data *payloads* though input and output *ports*. The processing modules are assembled in *pipelines*, using rigid rules of communication and composition.

The programming language used for the in-band partition was chosen to support maximum *efficiency*, *extensibilty*, and *portability*. Since the in-band partition should run with maximum efficiency, a programming language that compiles to efficient code should be used. Examples of such programming languages include C, C++, Objective C, and FORTRAN. Additionally, code in the in-band partition should be modular and extensible, to provide for extensibility and dynamic reconfigurability. Object-oriented languages like C++ and Objective C provide a good foundation for modularity and extensibility. The VuSystem was designed to be portable, and there are more implementations of C++ than Objective C, therefore the language chosen for the in-band processing partition was C++.

3.2.1 Modules

In-band VuSystem modules can be classified by the number of input and output ports they possess. The most common module classifications are sources, sinks, and filters.

- *Sources* are modules that possess one output port. Usually, a source module interfaces to an input device though the operating system. Modules that interface to video capture hardware are source modules.
- *Sinks* are modules that possess one input port. Usually, a sink module interfaces to an output device though the operating system. A module that interfaces to the window system presenting video frames on the screen is a sink module.
- *Filters* are modules that possess one input port and one output port. Usually, a filter module is used to perform any data conversion or condition detection. Filters can be implemented completely in software, or they can use special hardware to perform their function. A module that takes in uncompressed video frames and produces JPEG compressed video frames is a filter.

Modules with more than two ports can exist too. For example, a *multiplexer* module might have two input ports and one output port. Modules with three or more ports allow the construct of elaborate pipelines to perform complex tasks.

3.2.2 Payloads

VuSystem payloads are self-identifying, dynamically-typed objects that are logically passed between modules via ports. Examples of payloads include `VideoFrame` payloads, which contain a single uncompressed frame of video data, `AudioFragment` payloads, which contain a sequence of audio samples, and `Caption` payloads, which contain closed-caption text.

All in-band multimedia data in the VuSystem is represented as payloads to provide an abstract yet efficient handling mechanism for large amounts of multimedia data. Complicated implementation details such as shared-memory data regions and reference-counting pointer schemes are hidden behind the payload abstraction so that the designer of in-band processing modules needs only to know a few simple rules about payload handling. For example, it is always clear when a module gains ownership of a payload, and when it loses it.

Payloads are dynamically-typed in the VuSystem to provide for easy *data-directed* processing. Any VuSystem module with an input port can receive payloads of any type, and any module with an output port can send payloads of any type. This provides a flexibility in the composition of modules into processing pipelines that a statically-typed system cannot provide. For example, the standard VsFileSource composite module (page 106) includes in-line decompression modules that automatically convert compressed video frame payloads into decompressed video frame payloads, but pass transparently other payload types.

Each payload has two components: a *descriptor* and *data*. The *descriptor* component holds information about the entire payload, while the *data* component holds type-dependent media data. The descriptor component is implemented as a set of C++ class member functions. The data component is implemented as an opaque block of memory, whose precise representation is specified by the *descriptor* component.

A more detailed discussion of payloads can be found in Section 5.2, page 50.

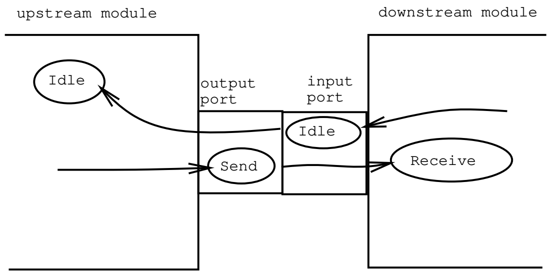


Figure 3.2: The VuSystem module data protocol.

3.2.3 The Module Data Protocol

The module data protocol is the mechanism used to transfer payload ownership between an *upstream* module and a *downstream* module within an application. Its principle features are:

- It does not require buffering between modules, which translates to reduced latency. Payloads are efficiently passed with one procedure call: a downstream module's `Receive` C++ class member function is called directly by an upstream output port's `Send` C++ class member function.
- It implements a *ready/not-ready* protocol that propagates timing constraints through back-pressure. By temporarily refusing a payload, a downstream module automatically throttles back upstream processing.
- It provides a cheap non-blocking scheduling mechanism that does not require multi-threading. If a downstream module has very little work to do, it can perform all its work in its `Receive` member function, otherwise it can schedule work to be done later.

Figure 3.2 shows the principle actors in the protocol and their relationships. A detailed discussion of the module data protocol can be found in Section 5.1 on page 49.

To pass a payload, the upstream module calls the `Send` member function on its output port, which calls the `Receive` member function of the downstream module. If the downstream module accepts the payload, it returns `True` from `Receive`, and the upstream module receives `True` from `Send`.

The downstream module could indicate it is not ready for more data by returning `False` from `Receive`. Upon receiving `False` from `Send`, the upstream module would stop trying to send data. Later, when the downstream module would become ready for more data, it would call the `Idle` member function on its input port, which would call the `Idle` member function on the upstream module. `Idle` on the upstream module would then send any waiting data to the downstream module.

3.2.4 Similarity to Streams

The architecture of the in-band partition is similar to that of the Unix Stream Input-Output System [35]. Streams provide a mechanism where a process can dynamically insert processing modules between processes and terminals or networks. Processing modules in streams are "pushed" onto a terminal or network, resulting in a conceptually vertical stack of processing modules, with the process at the top and the terminal or network at the bottom. Streams can also be used horizontally to provide a form of inter-process communication — processes can also connect directly to other programs through streams.

The VuSystem in-band processing partition is similar to streams in that both involve modules connected in complex pipelines, and both use data-driven run-to-completion scheduling of computation operations. The systems pass data and schedule operations differently: VuSystem processing modules use direct calls when passing data, while streams use data queues serviced by coroutines.

The most important difference between the VuSystem and streams is that VuSystem processing modules always run in user mode in a process address space. In streams, processing modules run in kernel mode in the kernel address space. In streams, privileged processes can install new processing modules in the kernel, but these modules run with maximum privileges and no memory protection. A bug in an in-band VuSystem module can only crash the application, but a bug in a streams module can crash the entire operating system. Streams modules are hard to debug, since their bugs have such severe consequences. They also must always be trustworthy, since they run with maximum privileges. VuSystem in-band processing modules can be debugged with ordinary user-mode debuggers, and can be run with low privileges without the danger of wreaking havoc on a computer system through system crashes. The VuSystem architecture is therefore more extensible than the streams architecture, because with it, any application can include special in-band processing modules that run safely in a user process.

3.3 Architecture Of The Out-Of-Band Partition

The nature of out-of-band processing is very different from in-band processing. For the out-of-band code, a programming system can be chosen that can handle user interfaces and other event-driven program functions well. When designing the out-of-band partition, programmability is more important than performance. For maximum ease of application development, the out-of-band partition of VuSystem is programmed in an interpreted scripting language. Application code written in this scripting language is responsible for creating and controlling the network of in-band media-processing modules, and controlling the graphical user-interface of the application.

3.3.1 The Scripting Language

The scripting language used in the VuSystem is the Tool Command Language, or Tcl [26], developed by John Ousterhout at the University of California at Berkeley. Tcl is designed as a simple but extensible command language. Its syntax is simple and concise enough that simple Tcl commands can just be typed in, but it is programmable and powerful enough that most of the control logic of a large application can be written in it. It has a simple and efficient interpreter, and a simple interface to C.

I chose Tcl over Common Lisp [28] or Scheme [22] because Tcl has a better interface to C and C++ and Tcl does not require a garbage collector. Tcl is designed from the start to have a simple and efficient interface to C. Scheme-to-C [32] is designed to have an efficient interface to C, but is not as good as Tcl in this respect. Through its interface to C, Tcl can easily support graphical user interfaces such as `TclXt` and `TclXaw`. Both Common Lisp and Scheme are still struggling to support standard graphical user interfaces. Finally, I could not afford to use a scripting language that required a non-incremental garbage collector. In-band processing is handled in the same thread as out-of-band processing, therefore a long period of time spent garbage collecting, while performing no media processing would be unacceptable.

The `TclXt` and `TclXaw` components of the VuSystem provide Tcl programming interfaces to the `Xt` and `Xaw` libraries respectively. These components enable the Tcl programmer to construct graphical user interfaces based on the Xt toolkit and the Athena widget set.

3.3.2 Communication Between In-Band and Out-Of-Band

Out-of-band Tcl scripts and in-band C++ modules communicate through *object commands* and *callbacks*:

- Out-of-band code is able to create and destroy in-band modules, query the state of in-band modules, and give commands to in-band modules, all through special Tcl *object commands* defined for each in-band module and port.
- In-band media-processing code signals out-of-band Tcl code whenever an appropriate in-band event occurs through Tcl *callbacks*.

The relationship of in-band code, out-of-band code, object commands, and callbacks is illustrated in Figure 3.1. A detailed discussion of object commands and callbacks can be found in Chapter 6.

Object commands are always completed in the in-band partition *synchronously* with the out-of-band requester: object commands execute immediately and completely when called from out-of-band scripts. In contrast, because of the time-critical nature of in-band code, it is unacceptable for in-band code to wait for a response from out-of-band code. Tcl *callbacks* are executed in the out-of-band partition *asynchronously* with the in-band partition: callbacks are only queued for execution when invoked from in-band code. Later, the VuSystem scheduler actually executes them. Since out-of-band callbacks do not execute immediately when they have been signalled by in-band code, they are only used to signal events to the out-of-band code, and cannot return values to their in-band signallers. Any in-band changes made by an out-of-band callback are performed through object commands.

3.4 The VuSystem Scheduler

The VuSystem scheduler provides scheduling services to in-band processing modules and the out-of-band script interpreter. It is designed to run all VuSystem code within one single-threaded Unix process. The VuSystem scheduler performs no preemption: module member functions called by the scheduler are run to completion before returning control to the scheduler. This simplifies the implementation of modules, since no locking or critical sections are required. No VuSystem code need be reentrant, so a variety of image processing and compression libraries can be incorporated into the VuSystem without qualification.

3.4.1 How Modules Interface to the VuSystem Scheduler

Modules interface to the VuSystem scheduler by arranging to have it call specific C++ class member functions at appropriate times:

- `Work` member functions are called to perform long computations.
- `Input` and `Output` member functions are called to perform input and output using Unix file descriptors.
- `Timeout` member functions are called to perform time-sensitive operations.

If a module needs to run a long computation, then it provides a `Work` member function that is occasionally called by the VuSystem scheduler. The module calls the `StartWork` and `StopWork` VuSystem scheduler interface precedures to start and stop the occasional calling of its `Work` member function. For example, the `VsPuzzle` module uses the `Work`

C++ class member function mechanism to do its scrambling of the video frame. A detailed discussion of `Work` member functions can be found in Section 5.6, page 56.

If a module does input or output on a Unix file descriptor, it provides an `Input` or `Output` member function that the system calls whenever the file descriptor is ready for input or output. The module uses the VuSystem scheduler interface procedures `StartInput`, `StopInput`, `StartOutput`, and `StopOutput` to control this service. For example, a file source module provides an `Input` member function for reading data from a file. A detailed discussion of `Input` and `Output` member functions can be found in Section 5.8, page 59.

If a module performs actions at precise times, it provides a `Timeout` member function that is called by the VuSystem scheduler after a time specified by the module has passed. The module schedules timeouts by calling the `StartTimeout` VuSystem scheduler interface procedure and cancels them by calling the `StopTimeout` VuSystem scheduler interface procedure. For example, a window sink module uses a timeout to present a video frame on the screen at a precise time. A detailed discussion of `Timeout` member functions can be found in Section 5.9, page 62.

Detailed discussions of VuSystem scheduler interface issues can be found throughout Chapter 5.

3.4.2 The Role of Payloads in VuSystem Scheduling

In the VuSystem, the assignment of computational resources to in-band processing modules is directly related to the flow of payloads. Modules schedule themselves when they receive input payloads. Starving a downstream module of payloads will reduce the amount of computational resources it will use. Similarly, a module that generates an output payload does not reschedule itself until after its payload has been accepted by its downstream module. Applying back-pressure to an upstream module reduces the amount of computational resources it will use.

Media payloads effectively act as scheduling *tokens*, automatically distributing computational resources to modules that need them. Through this interaction of scheduling and payload flow, VuSystem applications can automatically adapt to changing availability of computational resources. For example, a live video-processing application can automatically decrease the frame rate at which it is processing, as the load on the system increases and it gets a smaller fraction of the CPU time.

VuSystem source modules are designed to accept and adapt to back-pressure. For example, modules that interface to video capture devices skip frames when downstream modules refuse payloads. Similarly, sink modules are designed adapt to starvation. A module that presents video in a window leaves a previous frame in the window if starved for more frames.

I/O modules provide decoupling between the real world and internal processing modules through the conversion from open-loop protocols to the closed-loop module data protocol. Video capture modules, for example, convert from open-loop free-running 30 frame per second video to a sequence of video frame payloads delivered through the module data protocol. The modules use a frame buffer for this purpose. The frame buffer is filled on the open-loop side with a new video frame 30 times per second. On the closed-loop side, a video frame is extracted from the frame buffer only after a downstream module has accepted a previous frame from the video capture module. Video display modules use a similar approach to interface to display hardware: they update the workstation display video frame buffer with new video frames as they receive them.

This decoupling approach has its limits. For some I/O modules, adaptation to back-pressure and starvation is difficult or impossible to do well. For example, an audio sink module must maintain a continuous conversion of digital data to analog signals, or annoying audio pops and other distortions are introduced to the output. These modules define the temporal critical paths of VuSystem applications.

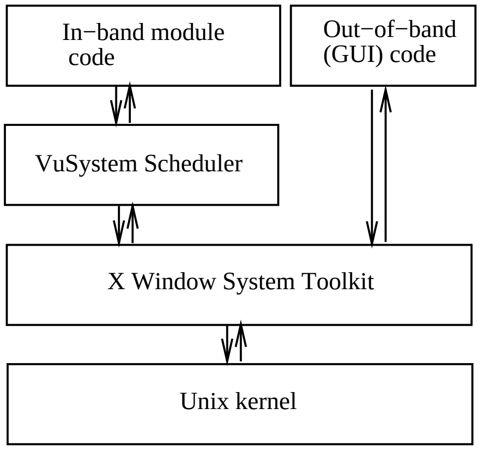


Figure 3.3: The VuSystem scheduler uses the X Window System Toolkit to provide scheduling support to in-band code.

3.4.3 Communication with the Unix Scheduler

VuSystem scheduler services are largely supported through the user-mode application scheduler provided by the X Window System Toolkit, or Xt [30] (Figure 3.3). The Xt library includes all the algorithms, data structures, and interfaces for scheduling input, output, timeout, and work procedures. The VuSystem scheduler gains leverage by being based on the Xt application scheduler: much code is shared, and VuSystem applications run with efficient combined scheduling of in-band media processing and out-of-band graphical user interface event handling.

VuSystem applications and the Unix process scheduler ultimately communicate through the `select` system call for BSD-based systems, and the `poll` system call for SVR4-based systems. The `select` and `poll` system calls provide the means for the VuSystem scheduler to specify the set of file descriptors for which it is waiting to become ready, and a timeout parameter to specify when it wants to wake up. `Input` and `Output` member functions are supported through the set of file descriptors, and `Timeout` member functions are supported through the timeout parameter. `Work` member functions are supported outside the `select` and `poll` system calls.

The scheduler was designed for maximum portability. It uses standard interfaces to Unix, and uses no real-time or threads extensions. With additional work, the VuSystem scheduler could be extended to make use of standard operating system interfaces that support concurrency through multi-threaded applications, such as those specified in POSIX.4 [14]. By using preemption and multiple threads of execution, the VuSystem would allow the concurrent execution of `Input`, `Output`, `Timeout` and `Work` procedures of different modules. Using this interface, VuSystem applications could make use of emerging multiprocessor systems by having threads of execution running concurrently on multiple processors.

3.5 Media Synchronization

Multimedia systems use some kind of synchronization mechanism to ensure that audio and video data simultaneously captured can also be simultaneously played back. To provide for the synchronous capture and playback of multimedia data in the VuSystem, I developed a system of payload timestamps:

- Media capture modules record the time of day at which media samples are captured in the `StartingTime` payload descriptor member of the payloads they create.
- File storage and retrieval modules preserve payload descriptors as well as data, so that timestamps are always available, even from stored payloads.
- A special `VsReTime` filter module (page 145) is used to perform operations on the `StartingTime` payload descriptor member of payloads.
- Media playback modules use the `StartingTime` payload descriptor member as the time of day for presentation of their input.

By using this system of payload timestamps, the VuSystem relies on the synchronized clocks of media capture and playback devices for synchronization during capture and playback.

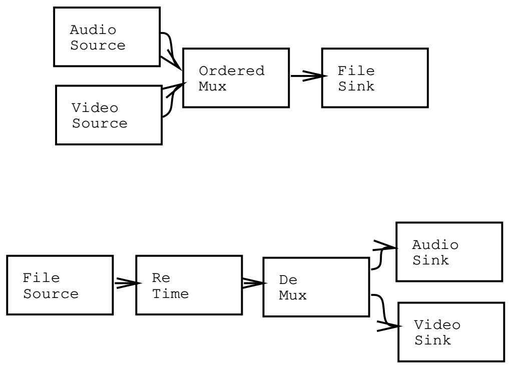


Figure 3.4: A network for synchronously capturing audio and video data and storing the data in a disk file, and a second network retrieving the data from the file, updating it with a re-timing filter, and synchronously playing it back.

Example

Figure 3.4 shows a network for synchronously capturing audio and video data, and storing the data in a disk file. A second network retrieving the data from the file, updating it with a re-timing filter, and synchronously playing it back is also shown.

During capture, the `AudioSource` and `VideoSource` modules record times of capture of media data in the descriptors of payloads they generate. The `OrderedMux` module merges the payload sequences into a single multiplexed sequence with ordered timestamps. Finally, the `FileSink` module saves payload descriptors and data in a disk file.

During retrieval, the `FileSource` module retrieves payload descriptors and data from a disk file. A `VsReTime` (page 145) module updates the timestamps stored in the payload desriptors, and a `DeMux` module restores the two sequences from the single multiplexed sequence. Finally, an `AudioSink` and `VideoSink` module present the media data at the times indicated by the updated timestamps in the payload descriptors.

3.5.1 The `VsRetime` Filter

The `VsReTime` filter module modifies the `StartingTime` payload descriptor member of payloads that pass through it. By adding a fixed offset to every timestamp, the filter allows the playback of media data at a time later than time of capture. The offset corresponds to the time difference between the time of the start of playback of a sequence, and the time of day of the start of capture of the sequence.

How It Works

Consider a sequence of media payloads, perhaps a sequence of video frames interleaved with corresponding audio fragments. When each video frame and audio fragment was captured by the VuSystem, the exact time of day of capture was recorded by the capturing *source* module in the `StartingTime` payload descriptor member for the payload. When the sequence is played back, the respective playback *sink* module presents the video frame or audio fragment at the time indicated by the `StartingTime` payload descriptor member. It is the job of the `VsReTime` module to change the `StartingTime` payload descriptor members so that the payload sequence can be played back correctly. It does so by keeping invariant the relative payload timestamps within the sequence:

$$T_{N,Playback} - T_{0,Playback} = T_{N,Capture} - T_{0,Capture} \tag{3.1}$$

The `VsReTime` module assigns the current time, plus a small offset to allow for some buffering before playback, to the `StartingTime` payload descriptor member for the first payload of the sequence:

$$T_{0,Playback} = CurrentTime + delay \tag{3.2}$$

For the rest of the payloads, `VsReTime` uses the time assigned to the first payload in the sequence and Equation 3.1:

$$T_{N,Playback} = T_{N,Capture} - T_{0,Capture} + T_{0,Playback} \tag{3.3}$$

The `VsRetime` filter can also cause the playback of stored media data at a speed different than its capture speed. To do this, the filter subtracts the timestamp of the first payload of the sequence, scales the result, and then adds the time of day of the start of playback of the sequence. Equation 3.1 can be rewritten to include a scale factor:

$$T_{N,Playback} - T_{0,Playback} = \frac{T_{N,Capture} - T_{0,Capture}}{speed} \tag{3.4}$$

Equation 3.3 can also be rewritten to include this scale factor:

$$T_{N,Playback} = \frac{T_{N,Capture} - T_{0,Capture}}{speed} + T_{0,Playback} \tag{3.5}$$

3.6 Implementation

The VuSystem was implemented on Unix workstations because of their overall high performance and ease of programming. Two specific workstation platforms were chosen: The Sun SPARCstation 10/512 and the Digital DEC 3000/400. On the Sun SPARCstation, the Sun VideoPix was used for video capture, and a standard X server for video output. Sun's audio hardware and software was used for audio input and output. On the Digital DEC 3000/400, the Vidboard [9], a LAN-based video capture subsystem, was used for video capture. The standard X server was used for video display, and DEC's audio hardware and AudioFile [10] software was used for audio input and output.

The toolkit is implemented as a program that works as a command shell, interpreting an extended version of the Tool Command Language (Tcl) [26]. Out-of-band code is written in Tcl, and in-band code is written in C++.

The toolkit includes a Tcl interface to the X Window System Toolkit and the Athena Widget Set for the graphical user-interface code. I chose to use the Xt intrinsics and the Athena widget set over the Tk [18] widget set provided with the Tcl distribution. The Xt intrinsics and Athena widget set were more robust and complete, and also had built-in features for scheduling in large applications. It was through the scheduling interface provided by the Xt intrinsics that I provided scheduling to the in-band modules.

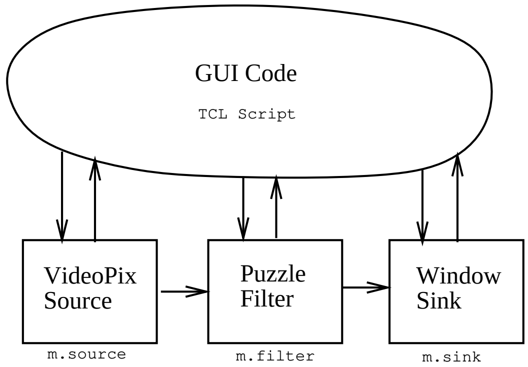


Figure 3.5: Block diagram of a rudimentary example application.

```
VsSunVfcSource $m.source \
  -scale $scale
VsPuzzle $m.puzzle \
  -dimension $dimension \
  -input "bind $m.source.output"
VsWindowSink $m.sink \
  -widget $w.screen \
  -input "bind $m.puzzle.output"

$m start
```

Figure 3.6: A fragment from the Tcl script of a rudimentary example application

Example

A video version of a 16-square puzzle can be built out of a video source module, a video sink module, and a filter module (Figure 3.5). This example application takes input from a camera or other video source, scrambles it, and then presents the output on a window. The Tcl script fragment that configures the modules and starts them running is shown in Figure 3.6. This fragment is not complete, but is meant to give an idea of what the scripts look like.

This script first creates an instance of a VsSunVfcSource module and names it `m.source`. The `-scale` parameter is provided to the module instance so that it will know at what resolution to capture video frames.

Next, the script creates an instance of the VsPuzzle module and names it `m.puzzle`. It passes a `-dimension` parameter to establish the number of rows and columns in the puzzle. It also instructs this module instance that its input port should be connected to the output port of the `m.source` module instance. The output port of m.source was automatically named `m.source.output`.

Next, the script creates a VsWindowSink module, specifying with the `-widget` option, that the widget named `w.screen` should be used to specify which window on the screen to use. The input port of `m.sink` is also specified to be connected to the output port of the `m.puzzle` module.

Finally the parent module instance named `m`, created before this script fragment was run, is given the `start` command, causing all its child module instances (with name `m.`*whatever*) to start.

3.7 Review

VuSystem applications are split into two partitions: one which does traditional *out-of-band* processing and one which does *in-band* processing. Out-of-band processing is that processing which performs the event-driven functions of a program. In-band processing is the processing performed on every video frame and audio fragment. In-band code is more elaborate in the VuSystem than in multimedia systems because VuSystem applications perform more analysis of their input media data.

In the VuSystem, the in-band processing partition is arranged into processing *modules* which logically pass dynamically-typed data *payloads* though input and output *ports*. These in-band modules can be classified by the number of input and output ports they possess. The most common module classifications are sources, with no input ports and one output port; sinks, with one input port and no output ports; and filters with one or more input ports and one or more output ports.

VuSystem payloads are self-identifying, dynamically-typed objects that are logically passed between modules via ports. Each payload has two components: a *descriptor* and *data*. The *descriptor* component holds information about the entire payload, while the *data* component holds type-dependent media data. The VuSystem relies on timestamps stored in payload descriptors and the synchronized clocks of media capture and playback devices for synchronization during capture and playback.

A *module data protocol* is used to transfer payload ownership between an *upstream* module and a *downstream* module within an application. It provides a *ready/not-ready* protocol that propagates timing constraints through back-pressure, does not require buffering between modules, and provides a cheap non-blocking scheduling mechanism that does not require multi-threading.

The out-of-band partition of the VuSystem is programmed in the Tool Command Language, or Tcl [26], an interpreted scripting language. Application code written in Tcl is responsible for creating and controlling the network of in-band media-processing modules, and controlling the graphical user-interface of the application. In-band modules are manipulated with *object commands*, and in-band events are handled with asynchronous *callbacks*.

Chapter 4

The VuSystem Application Environment

At the application level, the VuSystem is programmed in an interpreted scripting language. Application code written in this scripting language creates and controls the network of in-band media-processing modules, and controls the graphical user-interface of the application.

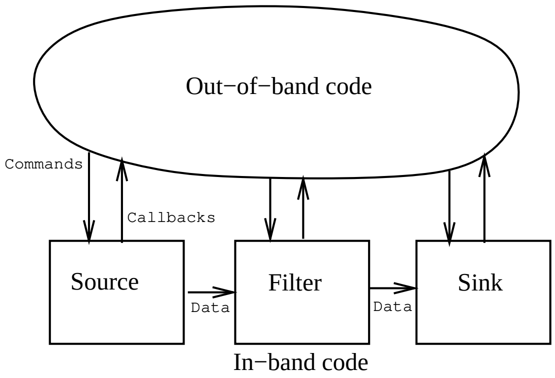


Figure 4.1: The structure of VuSystem applications.

Recall the VuSystem application structure diagram, shown again in Figure 4.1. The oval at the top of the diagram, labeled "Out-of-band code," corresponds to the application script, the "brains" of the program. This chapter describes this out-of-band partition. Reference information describing in-band VuSystem modules that can be manipulated by out-of-band code can be found in Appendix A, and general reference information useful to VuSystem application programmers can be found in Appendix B.

The out-of-band partition of VuSystem applications is programmed in Tcl, a simple but extensible interpreted scripting language. Tcl application scripts create and control a network of in-band media processing *modules* through *object commands*. Application scripts written in Tcl run in an *application shell*, creating the modules they need. They control the graphical user-interface of the application through a powerful and complete Tcl interface to the Xt intrinsics and the Athena widget set. By convention, application scripts are written to contain a *module creation procedure*, which creates the media processing and graphical user interface components; and a *main procedure*, which performs initializations and runs the application event loop.

4.1 The Tool Command Language

The scripting language used in the VuSystem is the *Tool Command Language*, or Tcl [26], developed by John Ousterhout at the University of California at Berkeley. It is an excellent programming language for assembling modules into flexible applications. Tcl is designed as a simple but extensible command language. Its syntax is simple and concise enough that simple Tcl commands can just be typed in, but it is programmable and powerful enough that most of the control logic of a large application can be written in it. It has a simple and efficient interpreter, and a simple interface to C.

Tcl syntax is similar to that of Unix shells, but it has additional Lisp-like constructs: Tcl uses curly braces to group elements, square brackets to invoke command substitution, and dollar signs to invoke variable substitution.

4.1.1 Tcl Data Types

Tcl has one data type: strings. All commands and values are strings. There is no other data type. It has no native representation of numbers or lists. All data is in the form of character strings. Even Tcl commands themselves are strings.

Since all data in the Tcl interpreter are strings, the embedding interface is simplified. It is easy to pass data between the Tcl interpreter and C code in an application. The C code need only be able to convert internal objects to and from strings. No library of converters between multiple representations is required.

Data types that can be easily represented in string form are quite natural to use within Tcl. For example, numbers can be easily converted to and from strings using standard mechanisms, and lists can be easily represented as strings, using curly braces for grouping. Data types too complex to be efficiently converted to and from strings can be represented in Tcl with *handles* or *object commands*.

Handles

Some objects of data types too complex to be efficiently converted to and from strings can be represented with string *handles*. Primitive commands that manipulate these objects use standard methods to convert string handles back to objects, and from objects to string handles.

A good example of objects that are represented by string handles are open files. The standard Tcl library provides commands to open, close, read and write files. These primitive use file *handles*, short strings that can be converted to and from file descriptors using hash tables.

Object Commands

Tcl *object commands* provide a powerful way to represent objects too complex to be efficiently converted to and from strings. For each object of a complicated data type, a unique Tcl command can be registered in the interpreter. Operations on an object can be performed by invoking its Tcl command, with the first argument to the command specifying the operation, and the rest of the arguments specifying the arguments to the operation. VuSystem object commands are implemented using Object Tcl [7], a dynamic object-oriented extension to Tcl developed for this purpose.

Object commands are used by the VuSystem to represent modules and ports. They provide a Tcl command name for each module and port, so that the module or port can be named and manipulated in Tcl. Each module and port is manipulated with its own object command. Each object command has several subcommands that allow the state of its object to be queried and changed. Each subcommand specifies a different operation that can be performed on the object. Object command names follow a hierarchical convention so that the possibility of name collisions is reduced.

Source	Function
`VsSunAudioSource`	Interfaces directly to the audio device on Sun workstations.
`VsAudioFileSource`	Interfaces to an AudioFile [10] server typically running on a Digital Alpha workstation.
`VsSunVfcSource`	Interfaces to the VideoPix video capture card on Sun workstations.
`VsXVideoSource`	Interfaces to video display adapters that have video capture capability through the XVideo X extension.
`VsVidboardSource`	Interfaces to the Vidboard [9], a LAN-based video capture subsystem.
`VsFileSource`	Interfaces to files stored using the native VuSystem file format.
`VsQtimeSource`	Interfaces to Quicktime [12] files.
`VsMpegSource`	Interfaces to MPEG [21] files.
`VsCaptionSource`	Interfaces to a closed-caption decoder through a serial line.
`VsExternalSource`	Assembles separate image files into a sequence of video frames.

Table 4.1: Some VuSystem sources.

The Tcl interface to the Xt toolkit and the Athena widget set also uses object commands to manipulate displays, application contexts, events, widgets, and widget classes. Widget resources and other object state can be manipulated through subcommands to these object commands.

4.2 Manipulating Modules

VuSystem media processing modules are created in Tcl with *creation* commands and manipulated with *object* commands. For each type of module that can be created, there is a creation command. For each module, there is an object command.

For example, the **`VsWindowSink`** Tcl command creates a **`VsWindowSink`** module. It also installs a new command in the Tcl interpreter to control the module. It takes as its first argument the name of the object command to create. The rest of the arguments to the creation command are parameters for the new module.

Object commands have many *subcommands*. The first argument to an object command is the name of the subcommand, and the rest of the arguments are parameters to the subcommand. This provides a form of object-oriented programming in Tcl. Invoking an object command is the same as sending a message to the module.

4.2.1 Module Types

VuSystem modules are perhaps best categorized by how many input and output ports they have. A module is either a *source*, a *sink*, a *filter*, or some *other* module. Descriptions of all the predefined VuSystem modules are available in Appendix A.

Sources

Modules with no input ports and one output port are called *sources* because they appear to the VuSystem to source data. Sources are typically I/O modules, since they interface to media capture devices or media storage systems. *Audio* sources interface to audio capture hardware. *Video* sources interface to video capture hardware. *File* sources interface to

Sink	Function
`VsSunAudioSink`	Interfaces directly to the audio device on Sun workstations.
`VsAudioFileSink`	Interfaces to an AudioFile [10] server typically running on a Digital Alpha workstation.
`VsWindowSink`	Interfaces to video display adapters through the X Window System.
`VsFileSink`	Interfaces to files stored using the native VuSystem file format.
`VsQtimeSink`	Interfaces to Quicktime [12] files.
`VsExternalSink`	Causes a sequence of video frames to be written to several image files.

Table 4.2: Some VuSystem sinks.

files. More exotic sources exist as well. Some sources provided with the VuSystem are listed in Table 4.1.

Sinks

Modules with one input port and no output ports are called *sinks* because they appear to the VuSystem to sink data. Sinks are typically I/O modules, since they interface to media playback devices or to media storage systems. *Audio* sinks interface to audio playback hardware. *Video* sinks interface to video playback hardware. *File* sinks interface to files. Some sinks provided with the VuSystem are listed in Table 4.2.

Filters

Modules with one input port and one output port are called *filters* because they are typically used to perform some signal processing operation on the data flowing through them. *Compression* filters compress or de-compress video frames. *Pixel format conversion* filters convert the format video frames. *Descriptor* filters perform operations on the descriptors of payloads. *Visual* processing filters perform various functions on video data. Some filters provided with the VuSystem are listed in Table 4.3.

Network modules are filters too. The `VsTcpClient` and `VsTcpServer` modules both have an input port and output port, so they are classified as filters, but they really act as endpoints of a TCP connection. Payloads that enter the input port of one of the modules come out of the output port of the other. The `VsTcpSource` and `VsTcpSink` modules need not be running in the same shell. They can be anywhere on the Internet. Through the use of these modules, and with the `VsTcpListener` module, networked VuSystem applications can be developed.

Other Modules

Modules with more than one input or output port provide mechanisms for splitting and merging payload sequences. Some modules with one input port and many output ports split a single timestamped sequence of payloads into multiple sequences. Some modules with many input ports and one output port merge multiple sequences into a single sequence. Modules subclassed from `VsEffect` and `VsGate2x1` combine payload sequences from two input ports to one output payload sequence based on operations on the contents of the data. Table 4.4 lists some VuSystem modules with more than one input and output port.

Filter	Function
VsJpegC	Compresses video frames using the JPEG [16] video compression standard.
VsJpegD	Decompresses video frames using the JPEG [16] video compression standard.
VsCCCC	Compresses 8-bit color video frames using a color cell compression algorithm.
VsCCCD	Decompresses 8-bit color video frames using a color cell compression algorithm.
VsQRLC	Compresses grayscale video frames using a simple run-length coding scheme.
VsQRLD	Decompresses grayscale video frames using a simple run-length coding scheme.
VsColor24to8	Converts 24-bit color to 8-bit color video frames.
VsColor8to24	Converts 8-bit color to 24-bit color video frames.
VsColor24toGray	Converts 8-bit color to grayscale video frames.
VsChannelSelect	Only passes payloads with certain channel identifiers.
VsChannelSet	Sets channel identifiers for `VsChannelSelect`.
VsReTime	Changes the starting-time and duration on payloads.
VsStepper	Allows explicit lockstep scripting control of each video frame passed.
VsWait	Waits for `VsFinish` payloads.
VsRateMeter	Measures the flow rate of payloads through it.
VvEdge	Performs edge processing on video frames.
VvThresh	Performs thresholding on pixel values.
VvHistogram	Converts a video frame into a pixel value histogram.
VsTcpClient	Interfaces to the client end of a TCP connection.
VsTcpServer	Interfaces to the server end of a TCP connection.

Table 4.3: Some VuSystem filters.

4.2.2 Object Subcommands

Each module object command has a set of *subcommands* that vary according to type of the module. The internal state of the module can be queried and changed with some of these subcommands. For example, the `VsVidboardSource` module has a `port` subcommand that is used to control from which input port it captures video. Table 4.5 shows a list of typical module subcommands.

4.2.3 Callbacks

Sometimes, out-of-band Tcl code in an application should be executed whenever an in-band event occurs. In this case, a *callback* is used. Many modules call their Tcl callback whenever a specific event occurs during in-band processing. For example, the `VsFileSource` module calls its callback when it reaches end-of-file.

Tcl callback commands are installed on each module with the `callback` module subcommand. Each module can have only one callback installed at a time. If a module can signal more than one type of event, it supplies keyword arguments to the callback command, so the command can determine which event occured. Table 4.6 shows a list of typical module callbacks.

4.3 The VuSystem Application Shell

The VuSystem is implemented as a Unix *application shell*: it is program that interprets an extended version of Tcl. Linked into the program are all standard in-band modules,

Module	Function
`VsDup`, `VsTap`	Copies payloads to provide identical output sequences.
`VsDeMux`	Splits its input sequence based on payload channel number.
`VsMerge`, `VsMux`	Merges multiple payload sequences into a single sequence.
`VsOrderedMux`	Merges multiple payload sequences into a single sequence, but ensures that timestamps in the output sequence are always monotonically increasing.
`VsFade`	Provides a fade effect between two sequences of video frames.

Table 4.4: Some VuSystem modules with more than one input and output.

Subcommand	Modules	Function
callback	All Modules	The callback.
children	All Modules	Lists all children in the structural hierarchy.
color	Video Sources	The color or grayscale toggle.
destroy	All Modules	Destroys module and all its children.
frameRate	Video Sources	The video frame rate to sample.
gain	Audio Sources and Sinks	The gain adjustment.
hue	Video Sources	The hue adjustment.
pathname	File Sources and Sinks	The pathname.
port	Many Sources and Sinks	The hardware input port to send output.
scale	Video Sources	The video frame size to generate.
start	All Modules	Starts module and all its children.
stop	All Modules	Stops module and all its children.

Table 4.5: Some VuSystem object subcommands.

implemented as C++ classes. Tcl scripts implement simple applications that use the default set of in-band modules. By linking additional in-band modules into the application shell, more complicated applications can be constructed.

The application shell defines the interface between the primitive module and command developer and the application script developer. At the primitive level, the module developer creates new primitive VuSystem modules and primitive Tcl commands and links them into the VuSystem application shell. At the application level, the developer writes Tcl code that runs in the application shell.

4.4 Programming The Graphical User Interface

The VuSystem includes facilities to construct an application graphical user interface with Tcl code. I developed *TclXt*, a Tcl interface to the Xt toolkit and *TclXaw*, a Tcl interface to the Athena widget set, to enable Tcl code to open X display connections, create windows, and install *callbacks* that cause Tcl commands to be executed on user input.

The `TclXt` and `TclXaw` components of the VuSystem provide Tcl programming interfaces to the `Xt` and `Xaw` libraries respectively. These components enable the Tcl programmer to construct graphical user interfaces based on the Xt toolkit and the Athena widget set. They provide an interface to the Athena widget set that is similar to `Tk` [18]:

- Widget instances are created by invoking a WidgetClass command. For

Condition	Modules	Indication
caption	VsWindowSink, VsCaptionSink	A caption was received.
compressRatio	Compression Filters	Filter is achieving this compression ratio.
detect	VsPayloadDetect	A payload of the specified type has been detected.
done	Effects Modules	The effect has completed.
rate	VsRateMeter	Payloads are passing at this rate.
sinkFinish	All Sinks	A `VsFinish` payload was received.
sinkStop	All Sinks	A `VsFinish` payload was received while in stopping mode.
solve	VsPuzzle	The puzzle has been solved.
sourceEnd	All Sources	The end of the source has been reached.
stepDone	VsStepper	A payload has been sent.
waiting	VsWait	A change of waiting status has occured.

Table 4.6: Some VuSystem callback conditions.

example, to create a button, one uses the `Command` Tcl command, which creates a `Command` widget.

- Initial values for widget resources are provided as keyword arguments to the creation command. The standard string conversion facilities provided by the `Xt` intrinsics convert the resource values from strings.
- Resources of existing widgets can be queried and changed through widget *subcommands*. Just as for the initial values of these resources, the standard string conversion facilities provided by the `Xt` intrinsics convert the resource values to and from strings.
- *Callbacks* and *Translations* can be specified as Tcl commands. These commands are executed whenever the given input event occurs.

`TclXt` and `TclXaw` provide all the ease of programming of the `Tk` widget set, but also provide an interface to the Athena widget set, a more standard set of widgets. Additional widgets based on the Athena widget set can trivially be added to the set of widgets that can be manipulated with Tcl. Motif widgets can be added too. Not only do these components provide a powerful interface to the Athena widget set, they also provide a *complete* interface. `TclXt` and `TclXaw` expose the entire interface to the `Xt` and `Xaw` libraries to the Tcl programmer. There is no library interface available to the C programmer that is not also available to the Tcl programmer. Detailed documentation on `TclXt` and `TclXaw` is available in Appendix D.

4.5 Application Scripts

The VuSystem provides considerable flexibility to the programmer of applications. This flexibility can result in widely varying coding styles. One programmer may not be able to understand the code of another. To enhance maintainability of VuSystem applications, some application script conventions have been established. One such convention is that all application scripts are written to contain *module creation* procedures and a *main* procedure.

Example

Figure 4.2 shows the graphical user interface for the **`vspuzzle`** program, an example VuSystem application. A simplified modular diagram of the application is shown in

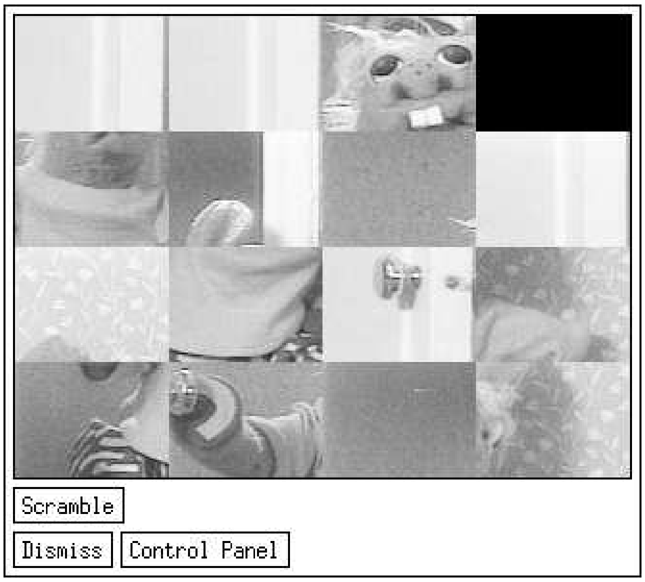


Figure 4.2: The graphical user interface of the **puzzle** application.

Figure 4.3.

4.5.1 The main Procedure

Every application script should include a **main** procedure. As in C programs, this **main** procedure should be the top level procedure for the application. At the very end of an application script should be the single Tcl command **main**, which causes the **main** procedure to be executed. Each **main** procedure performs five functions:

1. It initializes the graphical user interface libraries.
2. It extracts application-specific command-line arguments.
3. It initializes the run-time components of the VuSystem.
4. It creates its graphical user interface and media-processing modules.
5. It runs the application event loop.

Example

Figure 4.4 shows the code for the **main** procedure for the **vspuzzle** program. When it is called, the command line arguments to the application script lie in the global variable **argv**.

The first thing this **main** procedure does is to call the **xt appInitialize** (page 201) command to initialize the graphical user interface libraries. It then extracts its name and command arguments from the **argv** global variable and tells the top-level shell widget that it can be resized. Next, the run-time components of the VuSystem are initialized with a call to **vs appInitialize** (page 157).

A call to the **Puzzle** module creation procedure creates the graphical user interface components and the in-band VuSystem modules for the application. The user interface components are instantiated with the **realize** (page 227) Widget subcommand, and the in-band VuSystem modules are started with the **vs start** (page 164) command. Finally,

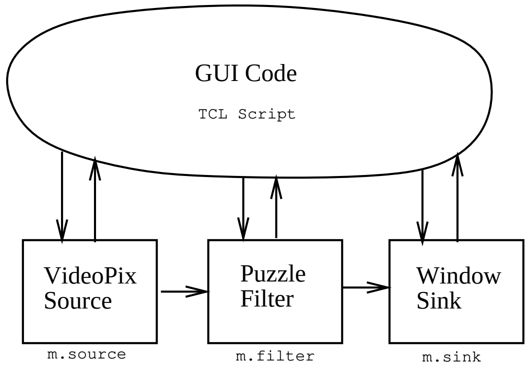


Figure 4.3: A block diagram of the `puzzle` application.

the `main procedure` runs the event loop of the application with `app_context mainLoop` (page 206). If the event loop ends with an error, the error is caught, a `VsErrorShell` is created with the error message, and the event loop is restarted.

4.5.2 Module Creation Procedures

If an application script creates a network of in-band processing modules, by convention it does it in a *module creation procedure*. Each module creation procedure does its work in three stages:

1. It extracts its parameters from the keyword-value pairs passed to it, and substitutes default values for parameters not supplied.
2. It creates its graphical user interface components.
3. It creates its in-band media-processing modules.

Module Creation Procedure Parameters

Module creation procedures are defined to have a calling sequence similar to that of graphical user interface components and VuSystem in-band modules:

- If a module creation procedure creates any graphical user interface components, the name of the object command for the top component in the structural hierarchy to be constructed is accepted as the first parameter.
- If a module creation procedure creates any in-band media processing modules, the name of the object command for the top module in the structural hierarchy to be constructed is accepted as the next parameter.
- All other parameters to module creation procedures are passed as keyword-value pairs.

Example

Figure 4.5 shows the code for the `Puzzle` example module creation procedure. This procedure creates the graphical user interface and in-band media processing modules for a simple video puzzle program. The graphical user interface is shown in Figure 4.2, and a diagram of the in-band media processing modules created is shown in Figure 4.6.

```
proc main {} {
  global argv errorInfo

  xt appInitialize appContext "Puzzle" argv {}

  set w [lindex $argv 0]
  set args [lrange $argv 1 end]
  $w setValues -allowShellResize true

  vs appInitialize appContext vs

  apply Puzzle $w.puzzle vs.puzzle \
    $args
  $w realize
  vs start

  while {[catch {appContext mainLoop} msg]} {
    VsErrorShell $w.err \
      -summary $msg \
      -detail $errorInfo
  }
}

main
```

Figure 4.4: The **main** procedure of the the **puzzle** application.

The first parameter, `w` is the object command name to use for the top component of the graphical user interface structural hierarchy. The second parameter, `m` is the object command name to use for the top component of the in-band media processing module structural hierarchy. The rest of the parameters are accepted in keyword-value pairs.

The `Puzzle` procedure first extracts the `dimension` parameter from the keyword argument list, using the `keyarg` (page 167) command. This parameter is used to establish the number of rows and columns in the video puzzle. If no `-dimension` keyword was supplied, a default value of 4 is substituted.

The `Puzzle` procedure next extracts the `scale` parameter from the keyword argument list. This parameter is used to establish the size of the video window in the video puzzle. A scale of 1 means full size, 2 means half size, and so on. If no `-scale` keyword was supplied, a default value of 2 is substituted.

Any other keyword parameters left in the args list are passed on to the top component of the graphical user interface hierarchy. The `keyargs` (page 167) command is used to remove the `-dimension` and `-scale` keyword arguments from `args`.

After parameter processing, the `Puzzle` procedure creates a graphical user interface structural hierarchy. At the top of the hierarchy a `Form` composite widget is created. All unrecognized parameters passed to the procedure are assumed to be widget subresource specifications, and so are passed to the `Form` widget creation command.

Inside the `Form` widget, the procedure creates a `VsScreen` widget. The size of the widget is determined by the `-scale` parameter, but it can also be resized by also passing `true` as a `-resizable` parameter.

Underneath the `VsScreen` widget, the procedure creates a button labeled `Scramble`. When pressed, this button will cause the `scramble` subcommand on the puzzle module to be called.

Underneath the `Scramble` button, the procedure creates a `Dismiss` button. When pressed, this button will destroy all the in-band media-processing modules and exit from the application.

Beside the `Dismiss` button, the procedure creates a `Control Panel` button. When pressed, this button will create a new top-level window which will contain control panels for the media-processing modules.

```
proc Puzzle {w m args} {
  set dimension [keyarg -dimension $args 4]
  set scale [keyarg -scale $args 2]
  set args [keyargs {-dimension -scale} $args exclude]

  apply Form $w \
    $args
  VsScreen $w.screen \
    -scale $scale \
    -resizable true
  Command $w.scramble \
    -label "Scramble" \
    -callback "$m.puzzle scramble" \
    -fromVert $w.screen
  Command $w.dismiss \
    -label "Dismiss" \
    -callback "catch {vs destroy}; exit" \
    -fromVert $w.scramble
  Command $w.controlPanel \
    -label "Control Panel" \
    -callback "VsPanelShell $w.controlPanel.shell -obj $m" \
    -fromVert $w.scramble \
    -fromHoriz $w.dismiss
  $w.screen overrideTranslations "<BtnDown>: tcl($m position)"

  VsEntity $m
  $m set w $w
  $m proc position {} {
    set clickpos [%event position]
    set width [$w.screen getValues -width]
    set height [$w.screen getValues -height]
    set x [expr {[lindex $clickpos 0]*[$self.puzzle dimension]/$width}]
    set y [expr {[lindex $clickpos 1]*[$self.puzzle dimension]/$height}]
    $self.puzzle position [list $x $y]
  }
  VsSunVfcSource $m.source \
    -scale $scale
  VsPuzzle $m.puzzle \
    -dimension $dimension \
    -input "bind $m.source.output"
  VsWindowSink $m.sink \
    -widget $w.screen \
    -input "bind $m.puzzle.output"
}
```

Figure 4.5: The `Puzzle` module creation procedure.

After creating all the graphical user-interface components, the `Puzzle` procedure adds a *translation* to the `VsScreen` widget. A translation is a mechanism supported by the X Window System Toolkit that maps a user-interface event to an application operation. This translation maps button presses, made in the window corresponding to the VsScreen widget, to a Tcl command "`$m position`", which causes the `position` subcommand to be invoked for the top component of the media processing structural hierarchy. In this way clicks on the video display are captured to change the position of the "hole" in the puzzle.

After creating the graphical user interface components, the `Puzzle` procedure creates an in-band media processing module structural hierarchy. At the top of the hierarchy a `VsEntity` module is created. This module does no media processing on its own, but exists as a common structural parent to the module network and holds common variables and methods.

On this `VsEntity` module, the `Puzzle` procedure creates an instance variable `w` which is the command name for the `Form` widget created earlier. The `Puzzle` procedure also defines a `position` method, which gets called whenever a mouse button is pressed inside

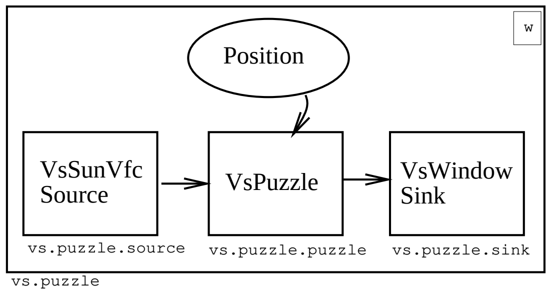


Figure 4.6: The VuSystem modules created by the Puzzle module creation procedure.

the `VsScreen` widget.

The `position` method is defined to turn each mouse click in a window into a command to the `VsPuzzle` in-band media-processing module in order to change the position of the "hole" in the puzzle. It gets the x and y coordinates from the mouse click, figures out in which square in the puzzle the click occured by comparing the position with the width and height of the `VsScreen` widget, and then calls the `position` method on the `VsPuzzle` module to move the hole to the square that was clicked.

Now the `VsPuzzle` procedure creates the in-band media-processing modules. It first creates a `VsSunVfcSource` module to capture video from a Sun Microsystems VideoPix video capture device. The `scale` parameter, previously used to establish the size of the display window, is also passed to the `VsSunVfcSource` command to establish the size of the video frames captured.

Next, the procedure creates a `VsPuzzle` module. The `dimension` parameter is passed to the `VsPuzzle` command to establish the number of rows and columns in the puzzle. The procedure also binds the input port of the module to the output port of the `VsSunVfcSource` module.

Finally, the `VsPuzzle` procedure creates a `VsWindowSink` module, which displays video frames on the computer screen. The name of the widget in which the frames should be displayed is supplied, so the module knows in which window on the screen the video frames should go. The procedure also binds the input port of the sink to the output port of the `VsPuzzle` module.

4.6 Review

The out-of-band partition of VuSystem applications is programmed in Tcl, a simple but extensible interpreted scripting language. In Tcl, all data is represented as strings, which simplifies the implementation of the language and its interface to the C++ code that implements the in-band partition. Data used in the in-band partition that cannot easily be converted to and from strings is represented by *handles* and *object commands*.

Tcl application scripts create and control a network of in-band media processing *modules* through object commands. The modules act as media *sources*, *sinks*, *filters*, and *other* modules, depending on the number of input and output ports they possess. Scripts use object *subcommands* to change the state of the modules, and modules use *callbacks* to signal back to scripts that in-band events have occurred.

A single executable image, the *application shell*, can implement many applications. All in-band processing modules are linked into the application shell. Application scripts written in Tcl run in the application shell, creating the modules they need. The application scripts can call on library scripts to perform common out-of-band operations.

Applications that require additional special-purpose in-band processing modules use *customized* application shells.

Application scripts control the graphical user-interface of the application, through *TclXt* and *TclXaw*, a powerful and complete Tcl interface to the Xt intrinsics and the Athena widget set. The interface presented by the toolkits is similar to that of `Tk` [18] but are to a more standard set of widgets.

By convention, application scripts are written to contain a *module creation procedure* and a *main procedure*. The module creation procedure creates all the in-band media processing and graphical user interface components for the application. The main procedure acts as the top-level procedure in the application, initializing the in-band partition and the graphical user interface, calling the module creation procedure, running the application event loop, and reporting errors to the user.

Chapter 5

Module Programming In The VuSystem

The in-band partition of a VuSystem application is structured as a reconfigurable directed graph of *modules*. The nodes of the graph are the in-band processing modules and the edges are associations of input and output *ports* on the modules. Through these ports logically pass *payloads* which hold the media data. A run-time system provides support to the modules for memory management, communication, and scheduling. In this chapter, I describe the implementation of modules, and discuss issues important to the module designer. Detailed VuSystem module development reference information can be found in Appendix C.

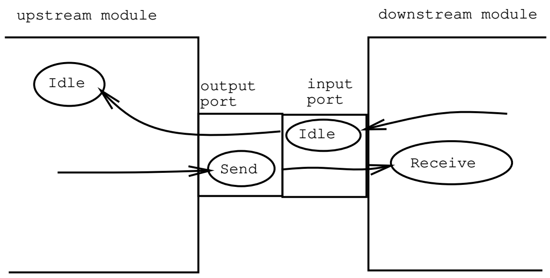


Figure 5.1: The module data protocol.

5.1 The Module Data Protocol

The module data protocol is the mechanism used to transfer payload ownership between an *upstream* module and a *downstream* module. Figure 5.1 shows the principal actors in the protocol and their relationships. To pass a payload, the upstream module calls the `Send` C++ class member function on its output port, which calls the `Receive` C++ class member function of the downstream module. If the downstream module accepts the payload, it returns `True` from `Receive`, and the upstream module receives `True` from `Send`. The upstream module then clears all internal references to the payload, since ownership of the payload has been passed.

Payload Type	Payload Description
`VsVideoFrame`	A single uncompressed full frame of video data.
`VsJpegFrame`	A single full frame of video data compressed according to the JPEG still image compression standard.
`VsQRLFrame`	A single full frame of grayscale video data compressed by quantizing and then run-length encoding.
`VsCCCFrame`	A single full frame of color video data compressed by a color-cell compression technique.
`VsAudioFragment`	A fragment of audio data, typically enough audio samples to cover roughly the same time that a a video frame would cover.
`VsCaption`	A single text caption, corresponding to closed captions transmitted on american television.
`VsStart`	Indication of the start of a payload sequence.
`VsFinish`	Indication of the end of a payload sequence.
`VsFlush`	Indication that modules should flush payloads from any buffers.

Table 5.1: The payload types.

If The Downstream Module Is Not Ready For More Data

The downstream module indicates it is not ready for more data by refusing a payload. It does this by returning `False` from `Receive` when called with a payload. Upon receiving `False` from `Send`, the upstream module keeps an internal reference to the payload, since ownership has not been passed. The upstream module may also note that the downstream module is not ready, and may stop trying to send data.

Later, when the downstream module is ready for more data, it calls the `Idle` C++ class member function on its input port, which calls the `Idle` C++ class member function on the upstream module. `Idle` of the upstream module notes that the downstream module is ready for data, and sends any waiting data to the downstream module. A downstream module may call `Idle` any time, even if it is not ready for more data, but if it ever refuses data by returning `False` from `Receive`, it *must* call `Idle` when it becomes ready for more data. Once the downstream module refuses a payload by returning `False` from `Receive`, the upstream module may assume the downstream module is not ready for more data until `Idle` is called.

If The Upstream Module Has No More Data

The upstream module can also stop sending data if it has no more payloads to send. If `Idle` is called and it has no data to send, it may *starve* the downstream module by not sending anything. Later, when the upstream module has more data to send, it calls `Send` with the payload. The upstream module may call `Send` any time, even if the downstream module has indicated that it is not ready for more data, but if it starves the downstream module, the upstream module *must* then call `Send` when it has more data. It should not wait for `Idle` to be called — once starved, a downstream module might not ever call `Idle` again.

5.2 Payloads

VuSystem payloads are self-identifying, dynamically-typed data structures which logically are passed through ports between modules. Examples of payloads include `VsVideoFrame` payloads, which contain single uncompressed full frames of video data;

Payload Type	Descriptor Component Contents
all payloads	Channel number, starting time, and active duration.
`VsVideoFrame`	Width, height, depth, bytes per line, bits per pixel, byte order, and pixel encoding.
`VsJpegFrame`	Width, height, quality, and original pixel encoding.
`VsQRLFrame`	Width, height, bytes per line, and quality.
`VsCCCFrame`	Width, height, and bytes per line.
`VsAudioFragment`	Samples per second, sample encoding, bits per sample, number of channels and byte order.
`VsCaption`	Length of the caption text.
`VsStart`	
`VsFinish`	
`VsFlush`	

Table 5.2: Descriptor component contents specific to payload type.

and `VsAudioFragment` payloads, which contain fragments of audio data. Table 5.1 shows the currently defined payload types.

Payloads are composed of two components: a *descriptor* component and a *data* component. The *data* component of a payload holds the media data of a payload, while the *descriptor* component holds information about the data. The descriptor component of a payload is represented as C++ class member variables. The precise representation of the data component of a payload is specified by the descriptor component.

5.2.1 Descriptor Components

Descriptor payload components hold general information for every payload as well as type-specific parameters. Table 5.2 shows descriptor component contents specific to payload type. Payload descriptor members common to all payload types include `Channel`, `StartingTime`, and `Duration`.

The `Channel` Payload Descriptor Member

The `Channel` payload descriptor member is a 32-bit integer used for multiplexing payloads. The `VsMux` and `VsOrderedMux` modules combine multiple input sequences of payloads into a single output sequence. They encode information in the channel number of the payloads to be used by `VsDeMux` to reconstruct multiple output sequences from a single input sequence.

A `VsMux` or `VsOrderedMux` module with n input ports records which input port from which payloads came, by updating the `Channel` payload descriptor so that `Channel` mod n returns which input port from which the payload came, and `Channel`/n returns the original `Channel` descriptor. A `VsDeMux` module with n output ports uses `Channel` mod n to select which output port to direct a payload, and replaces the `Channel` payload descriptor component with `Channel`/n.

Since the storage for the `Channel` payload descriptor is an integer of limited size, there is some limit to the depth of multiplexing that can be supported by the VuSystem. Being a 32-bit integer, the `Channel` payload descriptor can store up to 2^{32} possible encodings. This is enough to support 2-port multiplexers nested to a depth of 32, 3-port multiplexers nested to a depth of 20, 4-port multiplexers nested to a depth of 16, and so on. Since these are quite deep nestings of multiplexers, a fixed `Channel` value of 32 bits should be adequate for all multiplexer configurations in any forseeable VuSystem application.

Payload Type	Data Component Contents
VsVideoFrame	Image data for the video frame.
VsJpegFrame	Compressed image data for the video frame.
VsQRLFrame	Compressed image data for the video frame.
VsCCCFrame	Compressed image data for the video frame.
VsAudioFragment	Audio samples.
VsCaption	Caption text.
VsStart	
VsFinish	
VsFlush	

Table 5.3: Data component contents specific to payload type.

The `StartingTime` Payload Descriptor Member

The `StartingTime` payload descriptor member is a 64-bit time value which indicates the time at which a payload is valid. Media capture modules such as the `VsVidboardSource` and `VsAudioFileSource` modules record the time at which media samples are captured in the `StartingTime` payload descriptor member of the payloads they create. Media display modules such as the `VsWindowSink` and `VsAudioFileSink` modules display media samples at the times indicated by the `StartingTime` payload descriptor member.

The `VsReTime` filter module (page 145) modifies the `StartingTime` payload descriptor member of payloads that pass through it to allow display of media data at a time later than capture by adding a fixed offset to every timestamp. This offset corresponds to the time difference between the time at the start of the display of a sequence and the `StartingTime` of the first payload of the sequence.

The `VsRetime` filter can also be used to allow display of stored media data at a different rate than capture by subtracting out the `StartingTime` of the first payload of a sequence from all payloads in the sequence, scaling the `StartingTime`, and then adding the time at the start of the display of the sequence.

The `Duration` Payload Descriptor Member

The `Duration` payload descriptor member is a 64-bit time value which indicates the duration in which a payload is valid. Media capture modules such as the `VsVidboardSource` and `VsAudioFileSource` modules record the duration of validity in the duration payload descriptor member of the payloads they create.

The `VsReTime` filter module modifies the duration payload descriptor member of payloads that pass through it when it is being used to allow display of stored media data at a different rate than it was captured. It scales the duration of each payload that passes through it.

5.2.2 Payload Memory-Management and Marshalling

Data payload components are located in shared memory segments to facilitate cheap local interprocess communication of payloads. For example, the data component of a `VsVideoFrame` payload holds the image data for the video frame. Since the image data are therefore located in a shared memory segment, the `VsWindowSink` module uses the `MIT-SHM` X extension to pass the image data to the local X server to display the video frame in a window on the workstation screen. This extension uses shared memory segments to pass image data instead of the X connection byte stream, substantially increasing performance by eliminating copying of the data.

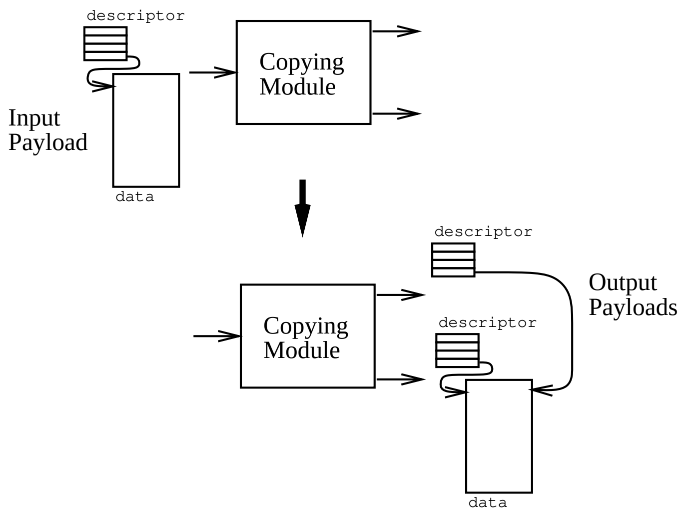


Figure 5.2: Making a shallow payload copy.

Shallow Copies And Deep Copies

Whenever any module is passed a pointer to a payload from an upstream module, it is considered to *own* the payload. It is responsible to either pass a pointer to the payload to a downstream module or to delete the payload. Once a module passes a payload pointer to a downstream module and the downstream module has accepted it, the upstream module no longer owns the payload. It should clear all pointers to the payload. It should not delete the payload, pass it on to another downstream module, or even look at it, since the downstream module may have already deleted it.

Modules can create copies of payloads to loosen these ownership restrictions. Modules can create *shallow* copies that share data components, or *deep* copies that have private data components. Deep copies of payloads are completely independent of each other, while shallow copies have independent descriptor components but shared data components.

Making shallow copies is much cheaper to do than making deep copies, since the data component isn't copied. Instead, data components are shared between shallow copies, and a reference counting pointer system is used to free the shared data when the last reference to it is deleted. The `VsDup` module creates $n-1$ shallow copies of its input payload and sends all n payloads to n downstream modules.

Side Effects And Shallow Copies

In general, a module cannot easily tell if its input payload has shallow copies, therefore it cannot easily tell if a data component of a payload is shared with other shallow copies of the payload. Therefore, it is important that modules not alter the data component of input payloads without first ensuring that the data component is not shared, as some other module might be processing a shallow copy of the payload with the shared data. Changing the data component of an input payload may cause unexpected side-effects. For example, an application might have created a shallow copy of a payload with the `VsDup` module, sending it to a filter, and sending the original payload to a module that displays it on the screen. If the filter modifies the data component of its input payload, unexpected results may appear on the screen.

A filter can ensure that the data component of its input payload is not shared

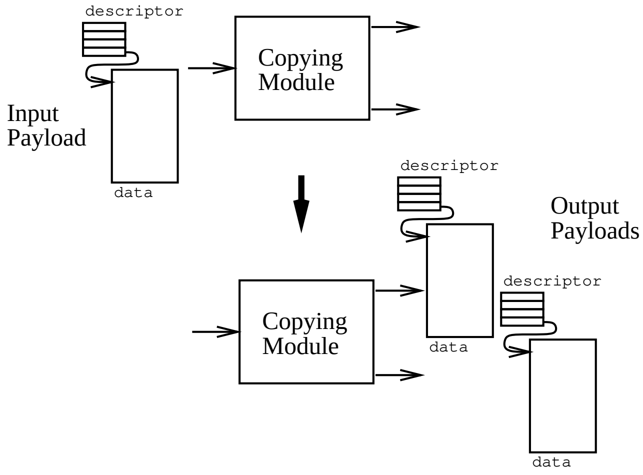


Figure 5.3: Making a deep payload copy.

by calling the `EnsureDataPrivate` payload C++ class member function. If the payload data is shared, `EnsureDataPrivate` replaces the data component with a copy, effectively transforming a shallow payload copy into a deep payload copy. If the payload data is not shared, `EnsureDataPrivate` does nothing. Use of shallow copies and `EnsureDataPrivate` effectively implements a *copy-in-write* policy for payload data components. It is efficient, since deep copies are made only when necessary.

Payload descriptors are never shared, so there is no restriction on modules changing the descriptors of payloads they own. For example, the `VsReTime` filter changes the starting time of payloads that pass through it, and the `VsChannelSet` filter changes the channel of payloads that pass through it.

Payload Encoding and Decoding

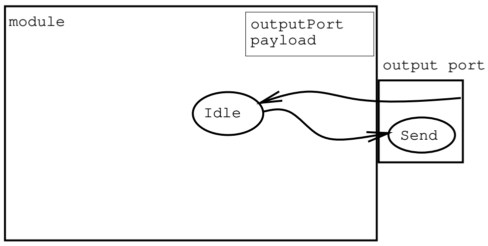
All payloads provide C++ class member functions to encode themselves into a reliable byte stream, and to decode themselves from a byte stream. By using these C++ class member functions, any sequence of payloads can be converted into a reliable byte stream and either be saved into a file or transmitted over a network.

5.3 Sending Data To A Downstream Module

The `Send` (page 171) VsOutputPort C++ class member function is used to send data to a downstream module, and an `Idle` (page 171) C++ class member function is implemented by the module designer to be called when a downstream module is ready for more data. `Send` can be called from anywhere in the upstream module, but by convention it is always called by `Idle`.

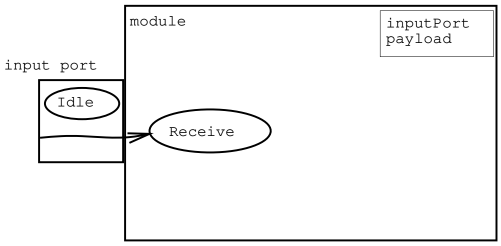
Example

Figure 5.4 shows a diagram and the code for the `Idle` C++ class member function of a simple source that sends data to a downstream module. In this example, the source module has two C++ class member variables: `outputPort`, a pointer to the output port for the module; and `payload`, which may point to a payload to be sent.




```
void
SimpleSource::Idle(VsOutputPort* op) {
  if (payload != 0 && op->Send(payload)) payload = 0;
}
```

Figure 5.4: A diagram of a simple source and the code for the `Idle` C++ class member function for the module.




```
Boolean
SimpleSink::Receive(VsInputPort*, VsPayload* p) {
  if (payload == 0) { payload = p; return True; }
  else return False;
}
```

Figure 5.5: A diagram of a simple sink and the code for the `Receive` C++ class member function for the module.

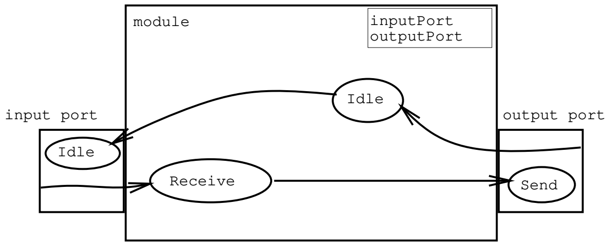
Here `Idle` takes one parameter: `op`, a pointer to the output port that is to receive more data. `Idle` checks `payload` to see if there is a payload to be sent. If there is a payload to be sent, it calls `Send` with the payload as a parameter. If the payload was accepted, `Send` returns `True` and `Idle` clears `payload`.

5.4 Receiving Data From An Upstream Module

The `Idle` (page 171) VsInputPort C++ class member function is used to indicate when a downstream module is ready to receive more data from an upstream module, and a `Receive` (page 172) C++ class member function is implemented by the module designer to accept the payload.

Example

Figure 5.5 shows a diagram and the code for the `Receive` C++ class member function of a simple sink that receives data from an upstream module. In this example, the sink module has two C++ class member variables: `inputPort`, a pointer to the input port for the module; and `payload`, which may point to a payload that was received.




```
Boolean
SimpleTransparentFilter::Receive(VsInputPort*, VsPayload* p) {
  return outputPort->Send(p);
}

void
SimpleTransparentFilter::Idle(VsOutputPort*) {
  inputPort->Idle();
}
```

Figure 5.6: A diagram of a simple transparent filter and the code for the `Receive` and `Idle` C++ class member functions for the module.

Here `Receive` takes two parameters: an ignored pointer to the input port from which the data arrives; and `p`, a pointer to the payload. `Receive` checks `payload` to see if it is null. If it is, `Receive` assigns `payload` to the payload, and returns `True` to accept the payload. If `payload` was not null, `Receive` rejects the payload by returning `False`. Later, the sink may clear `payload` and call `Idle` on the input port to indicate that the sink is to receive more data.

5.5 A Simple Transparent Filter

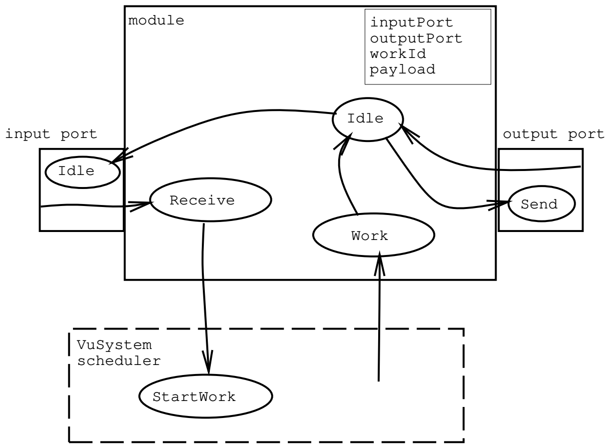
A simple transparent filter could be implemented which performs no data processing but instead passes all data transparently. Figure 5.6 shows a diagram and the code for the `Idle` and `Receive` C++ class member functions of a simple filter that transparently sends any data it receives. This is the most simple implementation of a filter. In this example, the filter module has at least two C++ class member variables: `inputPort`, a pointer to the input port for the module; and `outputPort`, a pointer to the output port for the module.

Here `Receive` takes two parameters: an ignored pointer to the input port from which the data arrives, and `p`, a pointer to the payload. `Receive` simply calls `Send` on the module output port with the payload, returning the result received to indicate whether or not the payload has been accepted.

Here `Idle` takes one parameter: an ignored pointer to the output port that is to receive more data. `Idle` simply calls `Idle` on the module input port, indicating that it is ready for more data.

5.6 Scheduling Computation Operations

Most filters do more than just pass their input payload on to the downstream module. They perform some *computation* based on the information stored in the payload. If this computation is trivial, it can be performed in `Receive`. More commonly, though, the computation takes some time, and should be performed in a procedure that can be scheduled to run at an appropriate time. Modules schedule nontrivial computation operations




```
Boolean
SimpleFilter::Receive(VsInputPort*, VsPayload* p) {
  if (payload == 0) { payload = p; workId = StartWork(); return True; }
  else return False;
}

Boolean
SimpleFilter::Work() {
  ...do the work...
  workId = 0; Idle(outputPort); return True;
}

void
SimpleFilter::Idle(VsOutputPort*) {
  if (payload != 0 && workId == 0 && outputPort->Send(payload)) payload = 0;
  if (payload == 0) inputPort->Idle();
}
```

Figure 5.7: A diagram of a simple filter and the code for the `Receive`, `Work`, and `Idle` C++ class member functions for the module.

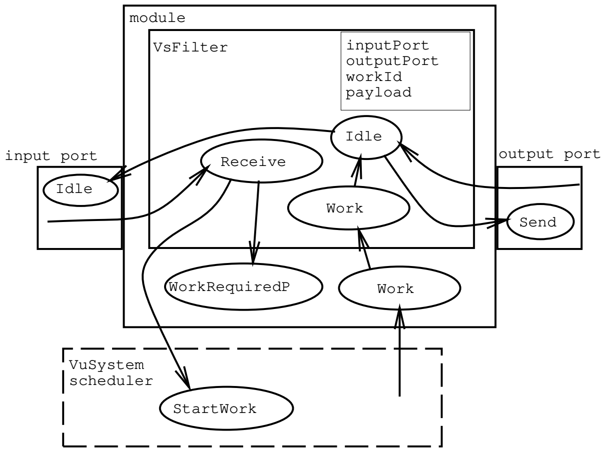
with `Work` (page 172) C++ class member functions. Once started with `StartWork` (page 172), `Work` is called regularly by the VuSystem scheduler until it either returns `True` or it is stopped with `StopWork` (page 172).

Example

The simple filter described in Section 5.5 could be modified to perform some computation on its payload in a `Work` C++ class member function. Figure 5.7 shows a diagram and the code for the `Receive`, `Work` and `Idle` C++ class member functions of a simple filter that processes data with `Work`. In this example, the filter module has at least four C++ class member variables: `inputPort`, a pointer to the input port for the module; `outputPort`, a pointer to the output port for the module; `workId`, a work identifier used to indicate `Work` been scheduled; and `payload`, a pointer to the payload being processed.

Here `Receive` takes two parameters: an ignored pointer to the input port from which the data arrives; and `p`, a pointer to the payload. `Receive` checks `payload` to see if it is null. If it is, `Receive` assigns `payload` to the payload, schedules the work function by calling `StartWork`, saves the result of `StartWork` in `workId`, and returns `True` to accept the payload. If `payload` was not null, `Receive` rejects the payload by returning `False`.

Here `Work` takes no parameters. After processing the payload, `Work` clears `workId` to indicate that is is finished. Just before returning `True` to indicate completion, `Work`




```
Boolean
StandardFilter::WorkRequiredP(VsPayload *p) {
  return VsVideoFrame::DerivePtr(p) != 0;
}

Boolean
StandardFilter::Work() {
  VsVideoFrame* frame = VsVideoFrame::DerivePtr(payload);
  caddr_t image_data = frame->Data().Ptr();
  VsXdrBlock newData(frame->Data().Fore());
  caddr_t compute_data = newData.Ptr();
  ...Perform the filter operation...
  frame->Data() = newData;
  return VsFilter::Work();
}
```

Figure 5.8: A diagram of a standard filter and the code for the `WorkRequiredP` and `Work` C++ class member functions for the module.

should call `Send` to send the processed payload downstream, and call `Idle` upstream if the payload was accepted, indicating the module is ready for more data. This is exactly what `Idle` does, so `Work` simply calls `Idle`.

Here `Idle` takes one parameter: an ignored pointer to the output port that is to receive more data. `Idle` check `payload` and `workId` to see if there is a payload to be sent, and that work in it has been completed. If there is a payload to be sent and work in it has been completed, `Idle` calls `Send` on the output port with the payload as a parameter. If the payload was accepted, `Send` returns `True` and `Idle` clears `payload`.

5.6.1 The Granularity of Scheduled Computation Operations

The VuSystem scheduler is non-preemptive: operations run to completion before control returns to the scheduler. It is also single threaded: only one operation runs at a time. This simplifies the design of modules in that no complicated locking or critical sections are required, since operations never overlap. On the other hand, it complicates the design of modules because no scheduled operation should run for too long.

Besides scheduling computation operations, the VuSystem scheduler schedules I/O operations and time-dependent operations. A long-running computation operation locks out any I/O or time-dependent operations while it is running. Because of this, scheduled computation operations should be designed to run with a fine enough granularity that the VuSystem scheduler can schedule more critical I/O and time-dependent operations.

It is up to the module developer to ensure that a scheduled computation operation does not run too long.

`Work` C++ class member functions should be written to run with a granularity appropriate for the applications in which they will be used. A good design rule is that a scheduled computation operation should take no more than 1/2 of a frame time. For example, for a VuSystem application that is expected to run at 15 frames/second, a computation operation should be designed to run no longer than 1/30 of a second. Applications built from modules following this rule exhibit sufficient levels of temporal sensitivity.

5.7 Standard Filters

Filter modules that simply perform computational transforms on data are called *standard* filters. They are written as subclasses of the `VsFilter` module class, and include a `WorkRequiredP` (page 180) C++ class member function and a `Work` (page 180) C++ class member function. `WorkRequiredP` is implemented by the module designer to return `True` if `Work` should be called for a given payload, and `False` if the payload should just be passed on without processing. `Work` is implemented by the module designer to perform the filter computation. When `Work` is called, the input to the computation is in `payload`. `Work` performs the computation and puts the result in `payload`. It calls `VsFilter::Work()` and returns its result.

Example

Figure 5.8 shows a diagram and the code for the `WorkRequiredP` and `Work` C++ class member functions of a standard filter. In this example, the filter module has at least four C++ class member variables: `inputPort`, a pointer to the input port for the module; `outputPort`, a pointer to the output port for the module; `workId`, a work identifier used to indicate that a call to `Work` has been scheduled; and `payload`, a pointer to the payload being processed.

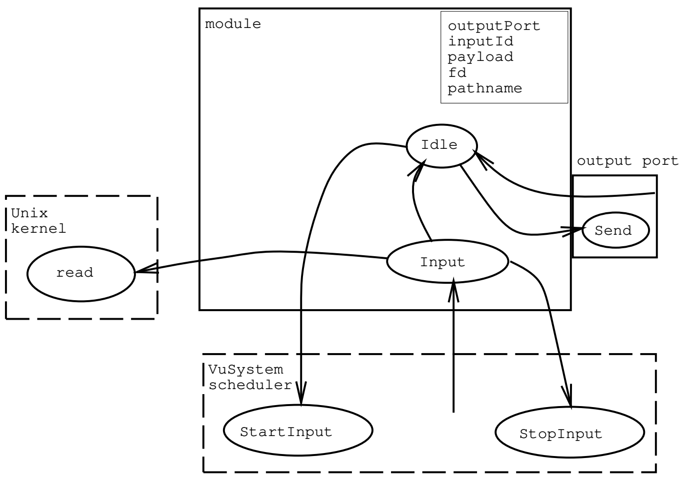
Here `Receive` takes two parameters: an ignored pointer to the input port from which the data arrives; and `p`, a pointer to the payload. `Receive` checks `payload` to see if it is null. If it is, `Receive` assigns `payload` to the payload, and returns `True` to accept the payload. If `payload` was not null, `Receive` rejects the payload by returning `False`. Later, some other C++ class member function of the sink may clear `payload` and call `Idle` on the input port to indicate that the sink is to receive more data.

5.8 Scheduling File I/O Operations

Some modules perform I/O, through Unix system calls, using Unix file descriptors. For example, a simple file source reads Unix files and generates payloads from the data in the files. If a module designer is not careful, the single threaded Unix process in which the VuSystem scheduler is running can become *blocked* by the Unix scheduler if an I/O operation is attempted by the module on an unready Unix file. In such a situation, since the entire process has been blocked, the VuSystem scheduler cannot allow some other module to run, even though one may be ready. To minimize I/O blocking of the VuSystem, modules that perform file input using Unix I/O use *nonblocking* I/O. Nonblocking I/O system calls return errors, instead of blocking, when files are not ready.

5.8.1 File Input

VuSystem modules that perform file input perform nonblocking I/O with the `Input` (page 174) C++ class member function. It is called by the VuSystem scheduler whenever a file indicated with `StartInput` (page 174) is ready for input. A typical `Input` C++ class




```
void
SimpleFileSource::Input(int, VsInputId) {
  ...read from the file, create payload when done...
  if (payload != 0) { StopInput(inputId); inputId = 0; Idle(outputPort); }
}

void
SimpleFileSource::Idle(VsOutputPort*) {
  if (payload != 0 && outputPort->Send(payload)) payload = 0;
  if (payload == 0 && inputId == 0 && fd >= 0) inputId = StartInput(fd);
}
```

Figure 5.9: A diagram of a simple file source and the code for the `Input` and `Idle` C++ class member functions for the module.

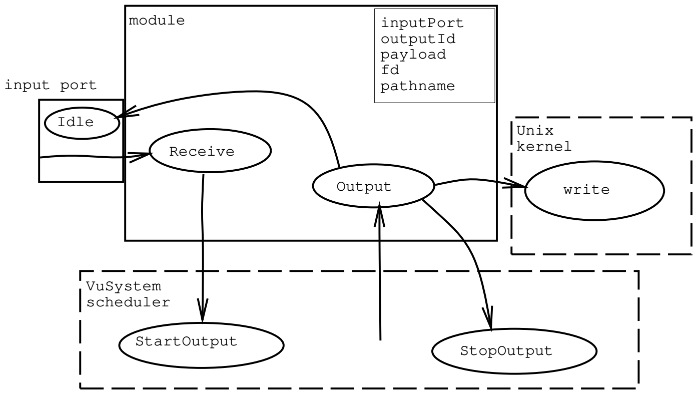
member function reads data from a file until a payload can be constructed or the file is no longer ready. When no more data is necessary from the file, `StopInput` (page 174) is used to stop calls to `Input` from the VuSystem scheduler.

Example

The simple source module described in Section 5.3 could be extended to read payloads from a file. Figure 5.9 shows a diagram and the code for the `Input` and `Idle` C++ class member function of a simple file source. In this example, the source module has at least four C++ class member variables: `outputPort`, a pointer to the output port for this module; `inputId`, an input identifier used to indicate `Input` has been scheduled; `payload`, which may point to a payload to be sent; `fd`, the Unix file descriptor for the file; and `pathname`, the name of the file.

Here `Input` takes two parameters: an ignored file descriptor that is ready for input; and an ignored input identifier. `Input` first reads any available data from the file. If enough data was read to build a payload, `payload` is set to point to the new payload. `Input` checks `payload` to see if it points to anything. If it does, `Input` stops scheduling file input operations with `StopInput` (page 174), clears `inputId` to indicate that `Input` is no longer scheduled, and calls `Idle` to try to send the new payload downstream to the next module.

Here `Idle` takes one parameter: an ignored pointer to the output port that is to receive more data. `Idle` checks `payload` to see if there is a payload to be sent. If there is a payload to be sent, it calls the `Send` function on the output port with the payload as a parameter. If the payload was accepted, `Send` returns `True`, so `Idle` clears `payload`.




```
Boolean
SimpleFileSink::Receive(VsInputPort*, VsPayload* p) {
  if (payload == 0 && fd >= 0) {
    payload = p; outputId = StartOutput(fd); return True;
  } else return False;
}

void
SimpleFileSink::Output(int, VsOutputId) {
  ...write to the file, delete payload when done...
  if (payload == 0) { StopOutput(outputId); outputId = 0; inputPort->Idle(); }
}
```

Figure 5.10: A diagram of a simple file sink and the code for the `Receive` and `Output` C++ class member functions for the module.

Finally, if `payload` is null, `inputId` is zero, and `fd` is a legal file descriptor, then `Idle` calls `StartInput` (page 174) with the file descriptor in order to cause `Input` to be called when input is available from the file. `Idle` sets `inputId` to the input identifier returned from `StartInput` to indicate that `Input` has been scheduled.

5.8.2 File Output

VuSystem modules that perform file output perform nonblocking I/O with an `Output` (page 175) C++ class member function. It is called by the VuSystem scheduler whenever a file indicated with `StartOutput` (page 175) is ready for output. A typical `Output` C++ class member function writes output data to a file until it has finished or the file is no longer ready. When no more data is necessary from the file, `StopOutput` (page 175) is used to stop calls to `Output` from the VuSystem scheduler.

Example

The simple sink module described in Section 5.4 could be extended to write payloads to a file. Figure 5.10 shows a diagram and the code for the `Receive` and `Output` C++ class member functions of a simple file sink. In this example, the sink module has at least four C++ class member variables: `inputPort`, a pointer to the input port for this module; `outputId`, an output identifier used to indicate `Output` has been scheduled; `payload`, which may point to a payload that was received; and `fd`, the Unix file descriptor for the file.

Here `Receive` takes two parameters: an ignored pointer to the input port from which the data arrives; and `p`, a pointer to the payload. `Receive` checks `payload` to see if it is null. If it is, and if `fd` is a legal file descriptor, then `Receive` assigns `payload` to the payload, calls `StartOutput` (page 175) with the file descriptor in order to cause `Output` to be called when the file is ready for output, saves the output identifier returned by `StartOutput` in `outputId` to indicate `Output` has been scheduled, and finally returns `True` to accept the payload. If `payload` was not null, `Receive` rejects the payload by returning `False`.

Here `Output` takes two parameters: an ignored file descriptor that is ready for output; and an ignored output identifier. `Output` first writes as much data as it can to the file. If enough data was written to complete the payload, the payload will be deleted and `payload` cleared. `Output` checks `payload` to see if it is null. If it is, `Output` stops scheduling file output operations with `StopOutput` (page 174), clears `outputId` to indicate that `Output` is no longer scheduled, and calls `Idle` on the input port to indicate that the sink is to receive more data.

5.8.3 The Granularity of Scheduled I/O Operations

As with computation operations, a long-running I/O operation locks out any other I/O and time-dependent operations while it is running. Because of this, scheduled I/O operations should be designed to run with a fine enough granularity that the VuSystem scheduler can schedule other I/O and time-dependent operations. It is up to the module developer to ensure that a scheduled I/O operation does not run too long.

`Input` and `Output` C++ class member functions should be written to run with a granularity appropriate for the applications in which they will be used. This is best achieved through the design of I/O operations that return when either their corresponding file becomes unready, or when they have completed one payload's worth of work.

5.9 Scheduling Time-Dependent Operations

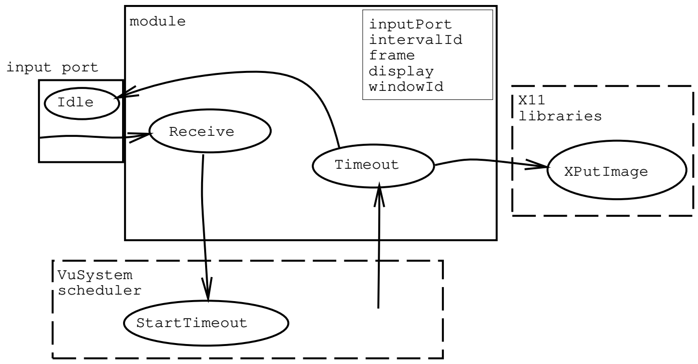
Since the VuSystem is designed to manipulate temporally sensitive data, some modules in the VuSystem need to perform operations at precise times. Typically these modules are sources or sinks that interface to media capture or display devices. For example, a window sink, which draws video frames on a window of a workstation display, needs to perform window updates at times specified by the video frames.

The VuSystem provides a mechanism in support of time-sensitive operation scheduling. If a module operation needs to be performed at a particular time, it is done in a `Timeout` (page 173) C++ class member function. It is called by the VuSystem scheduler after a time indicated through `StartTimeout` (page 173) has passed.

Example

The simple sink module described in Section 5.4 could be extended to write video frames on a window. This display operation would use the `Timeout` C++ class member function to ensure that the video frames are displayed at the right times. Figure 5.11 shows a diagram and the code for the `Receive` and `Timeout` C++ class member functions of a simple window sink. In this example, the sink module has at least six C++ class member variables: `inputPort`, a pointer to the input port for this module; `intervalId`, an interval identifier used to indicate `Timeout` has been scheduled; `frame`, which may point to a video frame that was received; `display`, a pointer to an X Window System display object; and `windowId`, an X Window System window identifier.

Here `Receive` takes two parameters: an ignored pointer to the input port from which the data arrives; and `p`, a pointer to the payload. `Receive` first calls `VsVideoFrame::DerivePtr` saving the result in the local variable `f` to check if the payload is a video frame. `VsVideoFrame::DerivePtr` returns a pointer to a video frame if the payload is a `VsVideoFrame` payload, and null if it is not.




```
Boolean
SimpleWindowSink::Receive(VsInputPort*, VsPayload* p) {
  VsVideoFrame* f = VsVideoFrame::DerivePtr(p);
  if (f != 0) {
    if (frame == 0) {
      frame = f;
      intervalId = StartTimeout(f->StartingTime());
      return True;
    } else return False;
  } else if (VsFlush::DerivePtr(p) != 0) {
    if (frame != 0) { delete frame; frame = 0; }
    if (intervalId != 0) { StopTimeout(intervalId); intervalId = 0; }
    delete p; return True;
  } else { delete p; return True; }
}

void
SimpleWindowSink::timeout(VsIntervalId) {
  ...display video frame...
  delete frame; frame = 0;
  intervalId = 0; inputPort->Idle();
}
```

Figure 5.11: A diagram of a simple window sink and the code for the `Receive` and `Timeout` C++ class member functions for the module.

If `f` is non-null (hence the payload is a `VsVideoFrame` payload), `Receive` checks whether `frame` already points to a video frame payload. If not, `Receive` assigns `frame` to the video frame, calls `StartTimeout` (page 173) with the timestamp of the video frame, saves the interval identifier in `intervalId` to indicate `Timeout` has been scheduled, and returns `True` to accept the payload. If `frame` already points to a frame, `Receive` returns `False` to decline the payload.

If the payload is not a video frame, `Receive` calls `VsFlush::DerivePtr` with the payload to check whether it is a `VsFlush` (page 193) payload. `VsFlush` payloads indicate that modules waiting for the appropriate time to pass before presenting data, should stop waiting and flush their data. If the payload is a `VsFlush` payload, `Receive` deletes any saved frame, cancels any scheduled time dependent operation with `StopTimeout` (page 173), deletes the payload, and returns `True` to accept it.

If the payload is neither a `VsVideoFrame` payload nor a `VsFlush` payload, `Receive` simply deletes the payload and returns `True` to accept it.

Here `Timeout` takes one parameter: an ignored interval identifier. It is called when the time saved in the `StartingTime` payload descriptor member of the video frame arrives. `Timeout` first displays the video frame, then deletes the video frame and clears `frame`. `Timeout` also clears `intervalId` to indicate that a time dependent operation is no longer scheduled. Finally, it calls `Idle` on the input port to indicate that the sink is to receive more data.

5.9.1 The Granularity of Scheduled Time-Dependent Operations

As with computation and I/O operations, a long-running time-dependent operation locks out any I/O and other time-dependent operations while it is running. Because of this, scheduled time-dependent operations should be designed to run with a fine enough granularity that the VuSystem scheduler can schedule I/O and other time-dependent operations. It is the responsibility of the module developer to ensure that a scheduled time-dependent operation does not run too long.

This responsibility is easily met through the following of a simple guideline. Fundamentally, `Timeout` C++ class member functions should be written to run with a granularity appropriate for the applications in which they will be used. This is best achieved through the design of time-dependent operations with constraints similar to that of computation operations: a scheduled time-dependent operation should take no more than 1/2 of a frame time.

5.9.2 The Precision of Scheduled Time-Dependent Operations

The VuSystem scheduler cannot preempt any running operation when a scheduled time-dependent operation comes due. Nor can it force the Unix scheduler to transfer control to the VuSystem application. Therefore the VuSystem scheduler cannot guarantee that a scheduled time-dependent operation runs precisely when its scheduled time passes. However, the scheduler can guarantee that a time-dependent operation will be scheduled immediately after its scheduled time passes.

This approach does not provide the precision that a true multi-threaded real-time scheduler could provide: precise scheduling of time-dependent operations through the preemption of less critical operations. Still, with well-designed modules, satisfactory application performance can be maintained with the VuSystem scheduler. Through control of the granularity of all scheduled VuSystem module operations, and through adjustment of the Unix process scheduler priority of the VuSystem application process, the scheduling precision of time-dependent operations can be controlled.

Modules that require time-dependent operations can be designed to have relaxed constraints on the precision of the scheduling of the operations. For example, an audio sink module that sends audio data to a kernel device driver for a speaker can use *buffering* to decouple the scheduling of its time dependent operations from the precise timing of the digital-to-analog conversion of the audio samples. By using buffering in a device driver, delay is introduced between the delivery of the data to the device driver and the actual audio output. This delay can be used by the sink module through the early scheduling of its time-dependent operations, to allow for late execution of its time-dependent operations. Using 1/8 to 1/4 of a second of buffering would provide ample time for late time-dependent operations.

Some applications cannot afford the latency introduced through buffering. For example, delays of more than a small fraction of a second can make a conferencing application unacceptably painful to use. In this situation, the precision of time-dependent operation scheduling can be improved through the sole use of modules that use fine granularity in all their scheduled operations. VuSystem applications with critical scheduling requirements can also be run with high Unix scheduler priorities (highly negative "nice" values), to encourage the Unix process scheduler to preempt other Unix processes when the VuSystem application needs to run.

Performance measurements reported in Chapter 7 demonstrate that this approach to the schedulling of time-dependent operations works reasonably well. A measurement made of the precision of `Timeout` calls shows most calls were made within one millisecond

```
void
SimpleFileSource::Start(Boolean mode) {
  if (fd >= 0 || inputId != 0 || payload != 0) Stop(True);
  if ((fd = open(pathname, O_RDONLY|O_NONBLOCK, 0)) < 0)
    VsError("%s: open %s: %s", Name(), pathname, strerror(errno));
 VsEntity::Start(mode);
 if (mode == False) Idle(outputPort);
}

void
SimpleFileSink::Stop(Boolean mode) {
  VsEntity::Stop(mode);
  if (inputId != 0) { StopInput(inputId); inputId = 0; }
  if (payload != 0) { delete payload; payload = 0; }
  if (fd >= 0) {
    if (close(fd))
      VsError("%s: close %s: %s",Name(),pathname,strerror(errno));
    fd = -1;
  }
  if (mode == False) {
    payload = new VsFinish(VsTimeval::Now(), 0);
    Idle(outputPort);
  }
}
```

Figure 5.12: The code for the `Start` and `Stop` C++ class member functions for a simple file source.

of their scheduled time, within the limit of the precision of the operation system clock. Of the calls not made within one millisecond, the vast majority were made within a few milliseconds. The measurements also show that precision gracefully degrades with increased system load.

5.10 Starting and Stopping

Some modules need to perform some operations at the beginning or end of in-band media processing. For example, a file source or sink module needs to open its file at the beginning of in-band media processing, and close its file at the end of in-band media processing. Modules that need to perform any processing at the beginning or at the end of in-band processing do so in `Start` (page 175) and `Stop` (page 175) C++ class member functions.

Example

The simple file sink module described in Section 5.8.1 would need to open the file at the start of in-band processing, and close the file at the end. Figure 5.12 shows the code for the `Start` and `Stop` C++ class member function for a simple file source. In this example, the source module has at least five C++ class member variables: `outputPort`, a pointer to the output port for the module; `inputId`, an input identifier used to indicate `Input` has been scheduled; `payload`, which may point to a payload to be sent; `fd`, the Unix file descriptor for the file; and `pathname`, the name of the file.

Here `Start` takes one parameter: `mode`, a Boolean value indicating whether `Start` should cause payloads to be sent downstream. `Start` first checks if `fd` is a legal file descriptor, which indicates that the file is already open. It also checks if `inputId` is non-null, which indicates that an input operation is scheduled, or if `payload` is non-null, which indicates that a payload is ready to be sent. If any of these conditions is true, `Start` calls `Stop` in abort mode to reset the module.

Here `Start` then attempts to open the file by calling `open`, storing the result in `fd`. If `open` fails, resulting in `fd` being negative, `Start` reports the error with `VsError` (page

```
SimpleSimpleFileSource::SimpleFileSource(Tcl_Interp* in, VsEntity* pr,
                                         const char* nm)
     :VsEntity(in,pr,nm),outputPort(new VsOutputPort(in,this,"output")),
     pathname(strcpy(new char[strlen("/dev/null")+1],"/dev/null")),
     fd(-1),payload(0),inputId(0)
{
  CreateOptionCommand("pathname", SimpleFileSourcePathnameCmd,
                      (ClientData)this, 0);
}

SimpleFileSource::~SimpleFileSource() {
  if (fd >= 0 || inputId != 0 || payload != 0) Stop(True);
  if (outputPort != 0) { delete outputPort; outputPort = 0; }
  if (pathname != 0) { delete pathname; pathname = 0; }
}
```

Figure 5.13: The code for the C++ class constructor and destructor of a simple file source.

181). `Start` calls `VsEntity::Start` in order to cause the `Start` C++ class member functions of any children of this module to be called. Finally, if `mode` is false, indicating that `Start` should try to send data to downstream modules, `Start` calls `Idle`, which will start an input operation.

`Stop` takes one parameter: `mode`, a Boolean value indicating whether `Stop` should cause `VsFinish` payloads to be sent, and whether filter and sink modules should wait for the `VsFinish` payloads before really stopping. `Stop` first calls `VsEntity::Stop` to cause the `Stop` C++ class member functions of any children of this module to be called. `Stop` checks `inputId`, and if it is non-zero, `Stop` then cancels any scheduled input operation with `StopInput` (page 174) and resets `inputId`.

`Stop` also checks the `payload` variable and if it is non-null, deletes the payload it is pointing to and clears the variable. If `fd` is a legal file descriptor, `Stop` calls `close` on it, reporting any errors with `VsError`. `Stop` resets `fd` to an illegal file descriptor, and if `mode` indicates that a `VsFinish` (page 192) payload should be sent, `Stop` sets `payload` to a new `VsFinish` payload and calls Idle, which will send the payload downstream.

5.11 Constructors and Destructors

Some module initializations cannot be performed by `Start`, and instead are performed by a C++ class constructor, which is executed by the C++ language system immediately after the module is first created. For example, input and output ports are created in the C++ class constructors of their parent module. Ports cannot be created at the start of in-band processing, since they must be connected together before the start of in-band processing. Similarly, some deinitializations cannot be performed by `Stop`, and instead are performed by the class destructor for the module, which is executed by the C++ language system immediately before the module is destroyed. For example, module input and output ports are deleted in module C++ class destructors, not at the completion of in-band processing.

Example

Figure 5.13 shows the code for the class constructor and class destructor for a simple file source. In this example, the source module has at least five C++ class member variables: `outputPort`, a pointer to the output port for the module; `inputId`, an input identifier; `payload`, a pointer to a payload; `fd`, the Unix file descriptor for the file; and `pathname`, to the name of the file.

```
VsEntity*
SimpleFileSource::Creator(Tcl_Interp* in, VsEntity* pr, const char* nm) {
  return new FileSource(in, pr, nm);
}

VsSymbol* SimpleFileSource::classSymbol;

void
SimpleFileSource::InitInterp(Tcl_Interp* in) {
  classSymbol = InitClass(in, Creator, "SimpleFileSource", "VsEntity");
}
```

Figure 5.14: The code for the `Creator` static C++ class member function, the `classSymbol` class variable, and the `InitInterp` static C++ class member function of a simple file source.

Here the class constructor takes three parameters: `in`, a pointer to a Tcl interpreter; `pr`, a pointer to the parent module; and `nm` the child name of the module. The constructor first calls the constructor for its parent class (`VsEntity`), passing on `in`, `pr`, and `nm`. It then creates its output port, saving a pointer to it in `outputPort`. The constructor sets its file name to `/dev/null`, and initializes `fd`, `payload`, and `inputId`. The constructor finally registers its `pathname` option subcommand with `CreateOptionCommand` (page 176).

Here the class destructor takes no parameters. If the module appears to be running, as indicated, by `fd`, `inputId`, or `payload` not reflecting their initial values, the destructor calls `Stop` (page 176) in abort mode. The destructor then destroys the modules output port if it has one, and finally destroys its file name string, if it has one.

5.12 Module Linkage Within The Application Shell

In order to make modules available to the application programmer, the modules have to be linked into the application shell. Tcl Commands to instantiate VuSystem modules are installed into the Tcl interpreter by calling an `InitInterp` static C++ class member function for each module.

Example

Figure 5.14 shows the code for the `Creator` static C++ class member function, `classSymbol` class variable, and `InitInterp` static C++ class member function of a simple file source. `Creator` returns a new instance of a `VsPuzzle` module, and `InitInterp` (page 179) calls `InitClass` (page 179), supplying its creator function, and saving the returned class symbol in a static variable. Since the VsPuzzle class was built on the VsFilter class, "VsFilter" is supplied as the superclass name to `InitClass`.

5.12.1 How The `InitInterp` Static C++ Class Member Function Is Called

In order to have module code linked into the application shell and registered in the Tcl command interpreter, the `InitInterp` static C++ class member function for the module (page 179) needs to be called by the application shell main program. The simplest way to have this done is to have it called by the `main` procedure of the application, right after all other Tcl interpreter initializations.

If a library of in-band modules is to be linked and registered, it is a good idea to provide a single `InitInterp` procedure for the whole library, which in turn calls the

```
#include <vs/vslib.h>
#include <vs/vsTcl.h>
#include <vsio/vsioInit.h>
#include <vv/vvInit.h>

int
main(int argc, char **argv) {
  Tcl_Interp *interp = Tcl_CreateInterp();
  VsInitInterp(interp);
  VsioInitInterp(interp);
  VvInitInterp(interp);
  SimpleFileSource::InitInterp(interp);
  return VsShellTopLevel(interp, argc, argv);
}
```

Figure 5.15: The code for the C++ `main` procedure for an application shell that includes the `SimpleFileSource` module.

`InitInterp` static C++ class member functions (page 179) for each module in the library. For example, the `main` procedure above calls `VsInitInterp`, `VsioInitInterp`, and `VvInitInterp`, which call the `InitInterp` static C++ class member functions for all modules in the `vs`, `vsio`, and `vv` libraries, respectively.

Example

Figure 5.15 shows the code for the `main` procedure for an application shell that includes a simple file source module. The `main` procedure takes two parameters: `argc`, the number arguments passed to the program; and `argv`, the arguments themselves. The `main` procedure first creates the Tcl interpreter and assigns the `interp` local variable to it. Next, the `VsInitInterp`, `VsioInitInterp`, and `VvInitInterp` procedures are called, to initialize the Tcl interpreter for the `vs`, `vsio`, and `vv` libraries, respectively. Then, the simple file source module class is installed in the interpreter with the `SimpleFileSource::InitInterp` procedure. Finally, the top-level processing loop is entered with a call to `VsShellTopLevel`, passing to it `argc` and `argv`, as well as `interp`.

5.13 Review

The in-band partition of a VuSystem application is structured as a reconfigurable directed graph of *modules*. The nodes of the graph are the in-band processing modules and the edges are associations of input and output *ports* on the modules. Through these ports logically pass *payloads* which hold the media data.

VuSystem payloads are self-identifying, dynamically-typed data structures which logically are passed through ports between modules. Payloads all have two components: a *data* component, which holds the media data; and a *descriptor* component, which holds information about the payload. Payload descriptor member variables common to all payload types include `Channel`, a 32-bit integer used for multiplexing payloads; `StartingTime`, a 64-bit time value which indicates the time at which a payload is valid; and `Duration`, a 64-bit time value which indicates the duration in which a payload is valid.

Media capture modules such as the `VsVidboardSource` and `VsAudioFileSource` modules record the time at which media samples are captured in the `StartingTime` payload descriptor member of each payload they create. Media display modules such as the `VsWindowSink` and `VsAudioFileSink` modules display media samples at the times indicated by the `StartingTime` payload descriptor member. The `VsReTime` filter module modifies the `StartingTime` payload descriptor member of each payload that passes through it. This provides for the display of media data at a time later than capture through the addition of a fixed offset to every timestamp. This offset corresponds to the time difference between the time at the start of the display of a sequence and the `StartingTime` of the first payload of the sequence.

Modules can create *shallow* copies of payloads that share data components, or *deep* copies that have private data components. Deep copies of payloads are completely independent of each other, while shallow copies have independent descriptor components but shared data components. Changing the data component of a payload with shallow copies may cause unexpected side-effects. Since a module cannot tell whether its input payload has shallow copies or not, it should not change the data component of its input payload.

The *module data protocol* is used to transfer payload ownership between an *upstream* module and a *downstream* module. To pass a payload, the upstream module calls the `Send` C++ class member function on its output port, which calls the `Receive` C++ class member function of the downstream module. If the downstream module accepts the payload, it returns `True` from `Receive`, and the upstream module receives `True` from `Send`. If the downstream module is not ready for more data, it returns `False` from `Receive` when called with a payload. Later, when the downstream module is ready for more data, it calls the `Idle` C++ class member function on its input port, which calls the `Idle` C++ class member function on the upstream module.

Most filters do their computation in a `Work` C++ class member function, which once started with `StartWork`, is called regularly by the VuSystem scheduler until it either returns `True` or is stopped with `StopWork`. *Standard* filters that simply perform computational transforms on data are written as subclasses of the `VsFilter` module class, and include a `WorkRequiredP` C++ class member function and a `Work` C++ class member function.

Modules that perform file input use an `Input` C++ class member function, which is called by the VuSystem scheduler whenever a file indicated with `StartInput` is ready for input. Modules that perform file output use an `Output` C++ class member function which is called by the VuSystem scheduler whenever a file indicated with `StartOutput` is ready for output. If an operation needs to be performed at a particular time, it is done in a `Timeout` C++ class member function, which is called by the VuSystem scheduler after a time specified with `StartTimeout` has passed.

Modules that need to perform any processing at the beginning or at the end of in-band processing use `Start` and `Stop` C++ class member functions. Module initializations that cannot be performed in a `Start` C++ class member function are put in the C++ class constructor for the module, and deinitializations that cannot be performed in a `Stop` C++ class member function are put in the C++ class destructor for the module.

In order to be available to the application programmer, modules have to be linked into the application shell. Tcl Commands to instantiate VuSystem modules are installed into the Tcl interpreter by calling an `InitInterp` static C++ class member function for each module.

Chapter 6

Communication Between In-Band And Out-Of-Band Partitions

Out-of-band processing is that processing which performs the event-driven functions of a program: the familiar event driven code typical of all interactive applications. *In-band* processing is the processing performed on every video frame and audio fragment: performed continuously on a running audio or video sequence. These two partitions are radically different in terms of functionality, programming language, and execution profile, yet they somehow need to cooperate to form the powerful application style used in the VuSystem. In this chapter, I discuss how they cooperate.

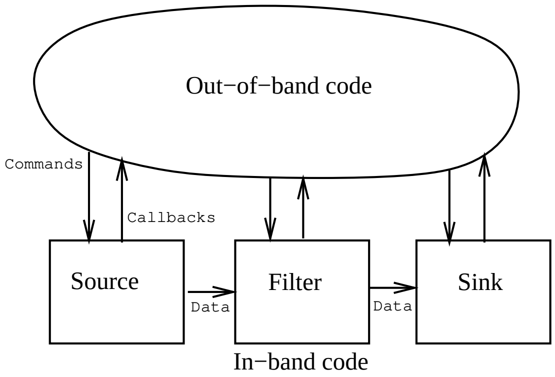


Figure 6.1: The structure of VuSystem applications.

Once again, recall the VuSystem application structure diagram, shown in Figure 6.1. The oval at the top of the diagram, labeled "Out-of-band code", corresponds to the application script, discussed in Chapter 4. The blocks at the bottom of the program, labeled "In-band code" correspond to the media processing modules, discussed in Chapter 5. The out-of-band partition uses *commands*, and the in-band partition uses *callbacks*.

The VuSystem uses what I call Tcl *object commands* to represent in-band modules and ports to out-of-band scripts. They provide a Tcl command name for each module and port, so that the module or port can be named and manipulated in Tcl. Each module and port is manipulated with its own object command. Each object command has several of what I call *subcommands* that allow the state of its object to be queried and changed. Each subcommand specifies a different operation that can be performed on the object.

Sometimes, out-of-band Tcl code in an application should be executed whenever an in-band event occurs. In that case, a *callback* is used.

6.1 Subcommands

Module subcommands are used to communicate from the out-of-band control partition to the in-band data partition. They are used to set parameters — adjusting virtual knobs and flipping virtual switches. Descriptions of the subcommands for all the predefined VuSystem modules are available in Appendix A.

Many subcommands are in the form of module *option* subcommands. Option subcommands are used to query and set values. They always return the current setting of a value, and take an optional argument to change the setting. Since they have a more restrictive form than other subcommands, option subcommands can be handled by graphical programming tools.

6.1.1 Subcommand Definition

Subcommands are normal Tcl command procedures, whose `ClientData` argument by default is a pointer to the associated module. Subcommands are declared C++ *friend* procedures to the module class, so they may manipulate private members of a module. They are declared to use C linkage.

Like all primitive Tcl command procedures, subcommands all have a standard form. Subcommands all take four parameters: an opaque client data parameter, a pointer to a Tcl interpreter, the number of command parameters, and the parameters themselves. A typical subcommand has three parts: input *parameter*, *processing*, and *return value*.

- The *parameter* part of a subcommand performs all input parameter processing. It first casts its opaque client data parameter to a pointer to its associated module. It checks the number of parameters, and uses the `VsTclErrArgCnt` (page 182) procedure to report incorrect parameter counts. It then parses its parameters, converting them to C data types. See Section C.13 for descriptions of procedures that provide for conversion of parameters to common C data types.
- The core of a typical subcommand is its *processing* part. This part does the actual work of the subcommand. Errors are caught in this part with `VsPushErrRec` (page 181), `VsPopErrRec` (page 181), and `VsErrRecToTclErr` (page 182).
- The *return value* part of a subcommand converts the C data type to a string, saves it in the Tcl interpreter's result slot, and returns the `TCL_OK` value. See Section C.14 for descriptions of procedures that provide for conversion of return values from common C data types.

Tcl subcommands are registered with the `CreateCommand` member function in the module class constructor. Option subcommands are registered using the `CreateOptionCommand` procedure in the module class constructor. See Section 5.11 for an example constructor.

Example

Figure 6.2 shows the code for the `SimpleFileSourceSourcePathnameCmd` Tcl subcommand procedure for a simple file source. In this example, the source module has at least two instance variables: `fd`, the Unix file descriptor for the file; and `pathname`, the name of the file.

```
int
SimpleFileSourceSourcePathnameCmd(ClientData cd, Tcl_Interp* in, int argc,
                                  char* argv[])
{
  SimpleFileSource* src = (SimpleFileSource*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?pathname?");
  if (argc > 1) {
    char* pathname;
    if (VsGetString(in, argv[1], &pathname) != TCL_OK)
      return TCL_ERROR;
    if (src->fd >= 0) {
      VsErrRec rec; VsPushErrRec(&rec);
      src->Stop(False);
      delete src->pathname;
      src->pathname = strcpy(new char[strlen(pathname)+1],pathname);
      src->Start(True);
      if (VsPopErrRec(&rec)) return VsErrRecToTclErr(in, &rec);
    } else {
      if (src->pathname != 0) delete src->pathname;
      src->pathname = strcpy(new char[strlen(pathname)+1],pathname);
    }
  }
  return VsReturnString(in, src->pathname, TCL_STATIC);
}
```

Figure 6.2: The code for the `pathname` subcommand for a simple file source.

The `SimpleFileSourceSourcePathnameCmd` Tcl subcommand procedure takes four parameters: `cd`, a Tcl client data parameter; `in`, a pointer to a Tcl interpreter, `argc`, the number of Tcl parameters to this command; and `argv`, the Tcl parameters. First, the procedure converts its `cd` parameter into a pointer to the SimpleFileSource module. It then checks the number of parameters. Since this command is an option command, one optional parameter is allowed. If the parameter count is wrong, the procedure signals a Tcl error by calling `VsTclErrArgCnt` (page 182) and returning the error code from it.

If `argc` is greater than 1, the procedure is called with a parameter, and this parameter is extracted into the `pathname` local variable, using `VsGetString` (page 187). The procedure checks if the current file is open and if `fd` is a legal file descriptor. If the file is open, the procedure stops the module by calling its `Stop` member function, deletes the old pathname, sets the `pathname` instance variable of the module with a new pathname string, and then restarts the module.

If any errors are reported by `VsError` during the execution of `Stop` and `Start`, they are caught in the VsErrRec `rec`, and are signalled as Tcl errors with `VsErrRecToTclErr` (page 182). If `fd` is not a legal file descriptor, which indicates the current file is not open, then the procedure simply deletes the old pathname string and replaces it with a new string.

Finally, `SimpleFileSourceSourcePathnameCmd` always returns the `pathname` instance variable of the module.

6.2 Callbacks

Modules process continuous sequences of in-band data, while out-of-band Tcl control processing deals with events. Modules turn continuous data into discrete events by calling Tcl *callbacks*. The VuSystem provides a facility for each module to have a callback. All modules have a `callback` option subcommand that is used to set the callback command string, and an `EvalCallback` (page 177) member function to call the callback. Descriptions of the callback conditions for all the predefined VuSystem modules are available in Appendix A.

```
void SimpleFileSource::Input(int, VsInputId) {
  ...
  if (...end of file...) EvalCallback("-sourceEnd 1");
  ...
}
```

Figure 6.3: The C++ code for the `Input` procedure of a simple file source that calls a callback.

Example

Figure 6.3 shows the code for the `Input` procedure of a simple file source that calls a callback. In this example, when the end-of-file condition is encountered by the source module, the callback command string is evaluated with the string `-sourceEnd 1` appended to it. This provides a Tcl application script with an indication that the simple file source has reached the end of its input file. An example Tcl script that would make use of the callback is shown in Figure 6.4.

```
proc sourceCallback {args} {
  set sourceEnd [keyarg -sourceEnd $args 0]
  if $sourceEnd {
    vs.source pathname "second.uv"
    vs.source callback ""
  }
}

SimpleFileSource vs.source \
  -pathname "first.uv" \
  -callback "sourceCallback"
```

Figure 6.4: How a simple file source callback might be used in Tcl.

6.2.1 Callback Definition

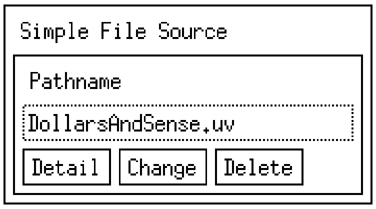
Callbacks are defined by the application programmer in Tcl, the application scripting language. Typically the Tcl application programmer provides a name of a Tcl procedure as the callback command. The Tcl procedure looks at its arguments to determine what event has occured.

Example

Figure 6.4 shows how a Tcl application programmer might make use of a simple file source callback that indicates end-of-file (Figure 6.3, page 74). This example code provides the automatic switching of the file source from file `first.uv` to the file `second.uv` when end-of-file is encountered on the first file.

The `sourceCallback` Tcl procedure takes a keyword argument list in its `args` parameter. It extracts any `sourceEnd` keyword parameter with the `keyarg` (page 167) command, defaulting to `0`. If `sourceEnd` is nonzero, `sourceCallback` changes the file for the source module using the `pathname` subcommand for the module. This will cause the source module to start on the file `second.uv`. The `sourceCallback` procedure also clears the callback for the source module using the `callback` subcommand for the module so that when the end of `second.uv` is signalled, the callback does not get run again.

After defining the `sourceCallback` procedure, a `SimpleFileSource` named `vs.source` is created, with its input file set to `first.uv` and its callback set to




```
SimpleFileSource classProc panel {w orient args} {
  apply Form $w \
    $args
  Label $w.label \
    -label "Simple File Source" \
    -borderWidth 0
  VsLabeledPathname $w.pathname \
    -label "Pathname" \
    -value [$self pathname] \
    -types {
      {"All Video Files" ".*\.(uv|rv|cv)"}
      {"Uncompressed Video Files" ".*\.uv"}
      {"Raw Video Files" ".*\.rv"}
      {"Compressed Video Files" ".*\.cv"}
      {"All Files" ".*"}
    } \
    -mustExist [true] \
    -callback "$self pathname" \
    -fromVert $w.label
}
```

Figure 6.5: A control panel for a simple file source and the code for the `panel` Tcl class procedure for the module.

`sourceCallback`. When started, `vs.source` will read from first.uv and evaluate the Tcl command string "`sourceCallback -sourceEnd 1`" when it encounters end-of-file.

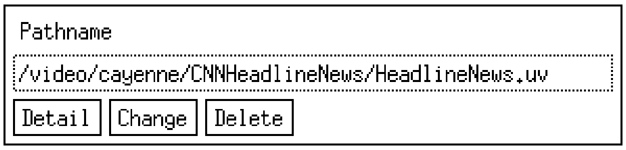
6.2.2 Tcl Callback Execution

Tcl Callbacks provide a mechanism for communication of events from the time-critical in-band partition of applications to the event-processing out-of-band partition. Since the VuSystem runs within one thread of control, immediate execution of out-of-band functions while more time-critical in-band functions wait would be unwise. Because of this, Tcl callbacks are executed *asynchronously*.

Instead of evaluating its Tcl command argument immediately, `EvalCallback` arranges for the Tcl command to be evaluated later, and returns immediately. The command is evaluated after more time critical in-band functions have been performed. Because Tcl callbacks are executed asynchronously, you cannot use `EvalCallback` to return a value from the Tcl command. You should use Tcl subcommands that can be called from your callbacks to perform control functions on the module.

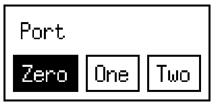
6.3 Control Panels

Module control panels provide a graphical user interface to option subcommands. Module control panels are defined with Tcl code. For every module that has a control panel, there is a library Tcl script that defines a `panel` class procedure for the module class. This `panel` class procedure contains code to construct a graphical user interface to the option subcommands of a module.




```
VsLabeledPathname $w.pathname \
  -label "Pathname" \
  -value [$self pathname] \
  -types {
    {"All Video Files" ".*\.(uv|rv|cv)"}
    {"Uncompressed Video Files" ".*\.uv"}
    {"Raw Video Files" ".*\.rv"}
    {"Compressed Video Files" ".*\.cv"}
    {"All Files" ".*"}
  } \
  -mustExist [true] \
  -callback "$self pathname" \
  -width 320 \
  -fromVert $w.form.caption
```

Figure 6.6: The user interface presented by, and the code for, the example use of the `VsLabeledPathname` control panel cliché.




```
VsLabeledChoice $w.form.port \
  -choices {{0 Zero} {1 One} {2 Two}} \
  -label "Port" \
  -value [$self port] \
  -callback "$self port" \
  -fromVert $w.form.label
```

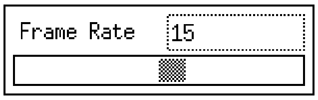
Figure 6.7: The user interface presented by, and the code for, an example use of the `VsLabeledChoice` control panel cliché.

Example

A control panel interface to the `SimpleFileSource` module might include a panel to allow the specification of `pathname` to the module. Figure 6.5 shows a control panel and the code for the `panel` Tcl procedure for a simple file source. In this example, the source module has at least one option subcommand: `pathname`, which specifies the name of the file.

The `panel` Tcl class procedure, like all `panel` class procedures, takes three parameters: `w`, the object command name for the main widget to be created; `orient`, set at either the word `-fromVert` or `-fromHoriz`, specifying the subpanels for module children that should be oriented horizontally or vertically; and `args`, the rest of the parameters to the procedure, which are supplied as a list suitable for processing with the `keyarg` (page 167) and `keyargs` (page 167) commands.

First, the `panel` Tcl class procedure creates a `Form` (page 201) widget as the main widget for the control panel. For the first panel member, the `panel` Tcl class procedure creates a `Label` (page 201) widget with no border and with the text "`Simple File Source`" to make a title. Then the procedure uses the `VsLabeledPathname` (page 196)




```
VsLabeledScrollbar $w.form.frameRate \
  -label "Frame Rate" \
  -value [$self frameRate] \
  -continuous [true] \
  -converter "vsLinearConverter 0 [vsDefault -frameRate]" \
  -inverter "vsLinearInverter 0 [vsDefault -frameRate]" \
  -callback "$self frameRate" \
  -valueWidth 70 \
  -width [expr {[vsDefault -frameRate]*5+10}] \
  -fromVert $w.form.hue
```

Figure 6.8: The user interface presented by, and the code for, an example use of the `VsLabeledScrollbar` control panel cliché.

procedure to define a panel member for selecting pathnames.

6.3.1 Control Panel Clichés

Control panel members usually can be built with a small set of code clichés. Some helper procedures are provided for constructing these control panel members.

- The `VsLabeledPathname` (page 196) procedure provides a user-interface cliché for displaying and changing file pathnames. For example, the `VsSunVfcSource` module uses the `VsLabeledPathname` procedure to define the interface to its `pathname` option subcommand, which is used to specify which Sun VideoPix device file to use. Figure 6.6 shows the user interface presented by, and the code for, an example use of the `VsLabeledPathname` control panel cliché.
- The `VsLabeledChoice` (page 196) procedure provides a user-interface cliché for displaying and changing parameters that are multiple-choice. For example, the `VsSunVfcSource` module uses the `VsLabeledChoice` procedure to define the interface to its `port` option subcommand, which is used to specify which analog video input port to select during video capture. Figure 6.7 shows the user interface presented by, and the code for, an example use of the `VsLabeledChoice` control panel cliché.
- The `VsLabeledScrollbar` (page 197) procedure provides a user-interface cliché for displaying and changing numeric parameters that vary over a range. For example, the `VsSunVfcSource` module uses the `VsLabeledScrollbar` procedure to define the interface to its `frameRate` option subcommand, which is used to specify the maximum frame rate to be used during video capture. Figure 6.8 shows the user interface presented by, and the code for, an example use of the `VsLabeledScrollbar` control panel cliché.

6.4 Review

Tcl *object commands* are used by the VuSystem to represent in-band modules and ports to out-of-band scripts. Each module is manipulated with its own object command. Object commands have several *subcommands* that allow the state of objects to be queried and changed.

A typical subcommand has three parts: an input *parameter* part which performs all input parameter processing; a *processing* part, which does the actual work of the subcommand; and a *return value* part, which converts the C data type to a string, saves it in the Tcl interpreter's result slot, and returns the `TCL_OK` value. Tcl subcommands are registered with the `CreateCommand` member function in the module class constructor.

Many subcommands are in the form of module *option* subcommands. Option subcommands are used to query and set values. They always return the current setting of a value, and take an optional argument to change the setting. Since they have a more restrictive form than other subcommands, option subcommands can be handled specially by graphical programming tools. Option subcommands are registered using the `CreateOptionCommand` procedure in the module class constructor.

Module control panels provide a graphical user interface to option subcommands. Module control panels are defined with Tcl code. For every module that has a control panel, there is a library Tcl script that defines a `panel` class procedure for the module class. This `panel` class procedure contains code to construct a graphical user interface to the option subcommands of a module. Control panel members usually can be built with a small set of code clichés. Some helper procedures are provided for constructing these control panel members.

Sometimes, out-of-band Tcl code in an application should be executed whenever an in-band event occurs. In this case, a *callback* is used. The VuSystem provides a facility for each module to have one callback. All modules have a `callback` option subcommand that is used to set the callback command string, and an `EvalCallback` member function to call the callback. Callbacks are defined by the application programmer in Tcl, the application scripting language. They are executed in the out-of-band partition, asynchronous with the in-band partition, and are only used to signal events to the out-of-band code.

Chapter 7

Performance

Performance of the in-band component of any media processing system is important. A useful system must meet perceptual-time constraints. The overhead of the run-time component of the system must be low. Any in-band processing modules within the system must be efficient. The scheduler used by the system must be able to perform operations at precise times. Finally, the system must have enough thoughput to support full-motion video.

In this chapter I report on experiments that verify that the VuSystem meets these performance requirements. I made five performance measurements on the VuSystem:

1. *Payload-passing overhead* was measured by measuring the amount of time a simple transparent filter takes to process a payload.
2. *Scheduler overhead* was measured by measuring the amount of time a filter that includes a `Work` C++ class member function takes to process a payload.
3. *Processing times of representative filter modules* were measured.
4. *Timeout precision* was measured by measuring the variation between the requested and the actual starting times of `Timeout` C++ class member functions.
5. *Total system throughput* was measured by measuring the amount of time a simple program takes to completely process a video frame.

7.1 Payload-Passing Overhead

To verify that the overhead of the module data protocol is low, I measured the amount of time a simple transparent filter takes to process a payload. I chose the `VsChannelSet` filter module (page 137). It simply sets the channel payload descriptor member of all payloads that pass through it, and does not look at the data of the payload. For this most simple filter module, the overhead of the module data protocol dominates the time taken to process a payload. Figure 7.1 shows a diagram of the module.

7.1.1 Experimental Setup

To measure the amount of time taken by the `VsChannelSet` filter to process a payload, a special test VuSystem program was written, and the throughput of the program was measured. Figure 7.2 shows the in-band modules used for the program.

In the program, the `VsTestVideoSource` (page 118) very cheaply generated a sequence of test payloads. These payloads were fed into N `VsChannelSet` filter modules

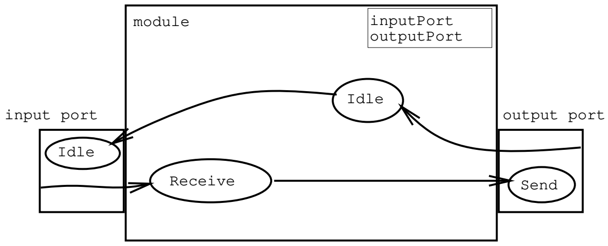


Figure 7.1: A diagram for the simple transparent filter module used for the data-passsing overhead measurement.

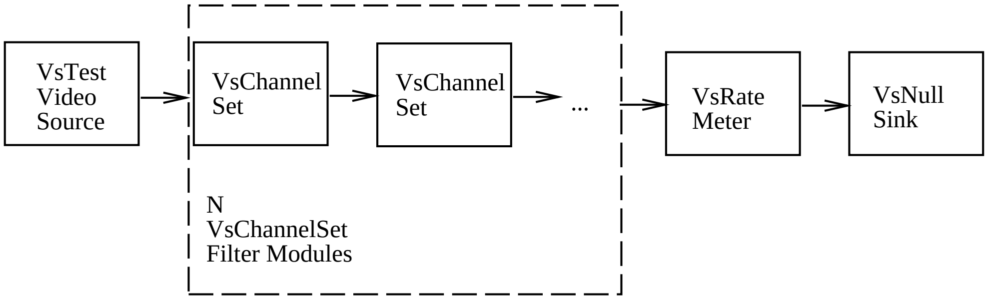


Figure 7.2: The payload-passing overhead experimental setup.

in series, where N was varied from 1 to 8192. The payloads then all traveled through a `VsRateMeter` filter module (page 144), and finally were deleted by a `VsNullSink` (page 134).

7.1.2 Determining Filter Processing Time

The amount of time to process a video frame in this test program is the sum of the processing time for each module,

$$T_{Program} = T_{VsTestVideoSource} + N \times T_{VsChannelSet} + T_{VsRateMeter} + T_{VsNullSink}. \quad (7.1)$$

Rearranging gives,

$$T_{VsChannelSet} = \frac{T_{Program}}{N} - \frac{T_{VsTestVideoSource} + T_{VsRateMeter} + T_{VsNullSink}}{N}. \quad (7.2)$$

So always,

$$T_{VsChannelSet} \leq \frac{T_{Program}}{N}. \quad (7.3)$$

Also, as N grows large, $T_{Program}$ also grows, but $T_{VsTestVideoSource}$, $T_{VsRateMeter}$, and $T_{VsNullSink}$ stay relatively constant. For N sufficiently large,

$$\frac{T_{Program}}{N} \gg \frac{T_{VsTestVideoSource} + T_{VsRateMeter} + T_{VsNullSink}}{N}. \quad (7.4)$$

So for sufficiently large N,

$$T_{VsChannelSet} \approx \frac{T_{Program}}{N}. \tag{7.5}$$

With the **VsRateMeter** module, we can get accurate measurements of the payload rate through the program ($R(N)$). The time taken by the program to process a payload ($T_{Program}$) is simply the reciprocal of the payload rate,

$$T_{Program} = \frac{1}{R(N)}. \tag{7.6}$$

Combining Approximation 7.5 and Equation 7.6,

$$T_{VsChannelSet} \approx \frac{1}{N \times R(N)}. \tag{7.7}$$

7.1.3 Results

Figure 7.3 shows $T_{VsChannelSet}$ as a function of N. Initially, as N grows, $T_{VsChannelSet}$ shrinks. With 500 filter modules in series, the time taken by **VsChannelSet** to process one payload was approximately 12 microseconds on the Sun SparcStation 10/512 and approximately 3 microseconds on the Digital DEC 3000/400. Given these numbers, we can conclude that the payload-passing overhead of the VuSystem was less than 12 microseconds on the Sun Sparcstation 10/512 and less than 3 microseconds on the Digital DEC 3000/400.

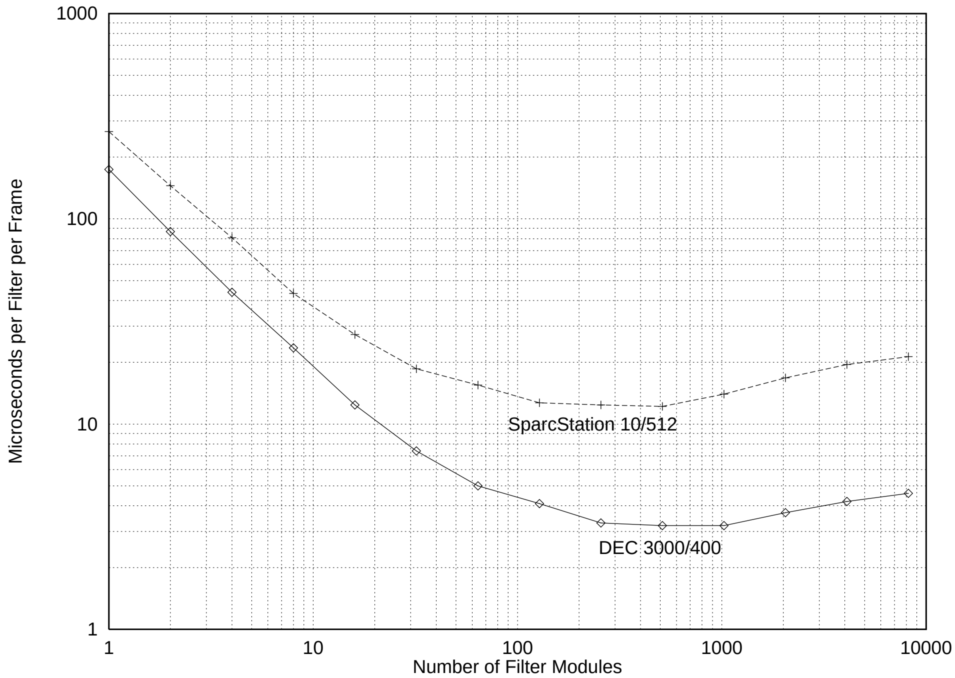


Figure 7.3: A plot of microseconds per filter as a function of the number of filter modules.

An interesting unexpected effect is also shown in the figure. Initially, as N grew from 1 to approximately 1000, the measured $T_{VsChannelSet}$ fell as predicted by the model. However, as the number of filter modules in series grows beyond 1000, the measured $T_{VsChannelSet}$ rises again. This effect is probably from memory system thrashing due to the large number of modules in use. These extremely long chains of transparent filter modules result in call stacks deep enough to exceed the size of the primary cache of the processors tested.

7.2 Scheduler Overhead

To verify the VuSystem run-time scheduler has low overhead, I measured the amount of time a mimimum filter with a `Work` C++ class member function takes to process a payload. I chose the `VsFilter` module. For each payload, it runs a `Work` C++ class member function which does nothing to the payload but pass it on. For this simple standard filter module, the overhead of scheduling `Work` dominates the time taken to process a payload. Figure 7.4 shows a diagram of the module.

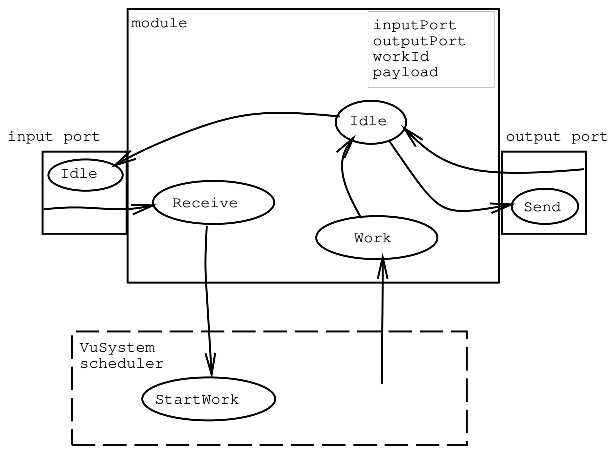


Figure 7.4: A diagram for the simple standard filter module used for the scheduler overhead measurement.

7.2.1 Experimental Setup

To measure the amount of time taken by the `VsFilter` module to process a payload, the special test VuSystem program described in Section 7.1 was modified to use the `VsFilter` module, and the throughput of the program was measured. Figure 7.5 shows the in-band modules used for the program.

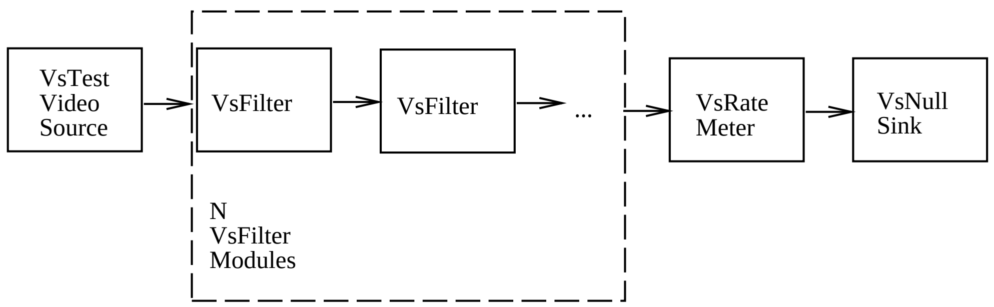


Figure 7.5: The scheduler overhead measurement experimental setup.

In the program, the `VsTestVideoSource` (page 118) very cheaply generated a sequence of test payloads. These payloads were fed into N `VsFilter` filter modules in series, where N was varied from 1 to 512. The payloads then all traveled through a `VsRateMeter` filter module (page 144), and finally were deleted by a `VsNullSink` (page 134).

7.2.2 Results

Figure 7.6 shows $T_{VsFilter}$ as a function of N. Initially, as N grows, $T_{VsFilter}$ shrinks. With 50 filter modules in series, the time taken by `VsFilter` to process one payload was approximately 150 microseconds on the Sun SparcStation 10/512 and approximately 115 microseconds on the Digital DEC 3000/400. Given these numbers, we can conclude that the scheduler overhead of the VuSystem for `Work` was less than 150 microseconds on the Sun Sparcstation 10/512 and less than 115 microseconds on the Digital DEC 3000/400.

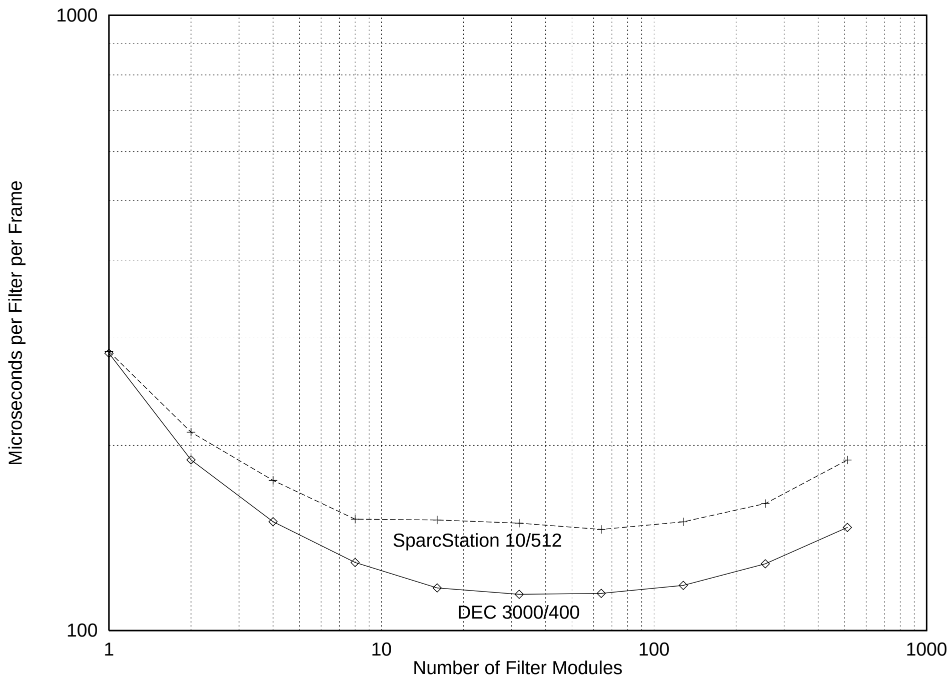


Figure 7.6: A plot of microseconds per filter as a function of the number of filter modules.

Just as for the payload-passing overhead measurements, the scheduler overhead rises when large number of modules are involved. Initially, as N grew from 1 to approximately 100, the measured $T_{VsFilter}$ fell as predicted by the model. However, as the number of filter modules in series grows beyond 100, the measured $T_{VsFilter}$ rises again. Once again, this effect is probably from memory system thrashing due to the large number of modules in use and `Work` C++ class member functions concurrently scheduled.

7.3 Processing Times Of Representative Filter Modules

To verify that media processing modules in the VuSystem are able to perform their functions with perceptual-time granularity, I measured the amount of time two representative filter modules took to process a video frame. I measured the processing times for the `VsPuzzle` filter module (page 141) and the `VvEdge` filter module.

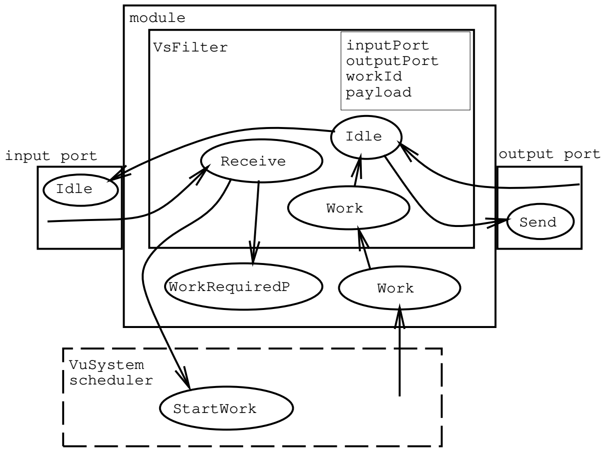


Figure 7.7: A diagram for the representative filter modules used for the processing times measurement.

The `VsPuzzle` filter module is the module used as an example throughout this report. It scrambles a video frame to form a video puzzle. The `VvEdge` filter module is part of Stasior's library of vision service modules [8]. It performs edge detection on video frames. Both of these modules are based on the `VsFilter` module, and contain `Work` C++ class member functions that perform a substantial amount of computation. Figure 7.7 shows a diagram of these modules.

7.3.1 Experimental Setup

The special test VuSystem program described in Sections 7.1 and 7.2 was modified to use one of the representative modules. Figure 7.8 shows the in-band modules used for the program.

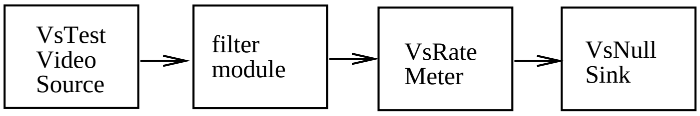


Figure 7.8: The setup used to measure the processing times of representative filter modules.

In the program, the `VsTestVideoSource` (page 118) very cheaply generated a sequence of test video frame payloads of various sizes. These payloads were fed into the filter module. The video frame payloads then traveled through a `VsRateMeter` filter module (page 144), and finally were deleted by a `VsNullSink` (page 134).

7.3.2 Determining Filter Processing Time

This configuration is identical to the one described in Sections 7.1 and 7.2, except $N = 1$, therefore the inequality 7.3 is still valid. In addition, because `Work` in the filter module does a lot of processing, the inequality 7.4 is valid, even though there is only one filter module being tested in this setup. Therefore for this setup,

$$T_{Filter} \approx T_{Program}. \tag{7.8}$$

Combining with Equation 7.6,

$$T_{Filter} \approx \frac{1}{R}. \tag{7.9}$$

where R is the frame rate reported by the `VsRateMeter` module.

7.3.3 Results

Figure 7.9 shows T_{Filter} as a function of the frame size for the input video frame. A frame size of 640x480 represents a full-sized video frame, 320x240 represents a half-size frame, and so forth. The number of pixels in each video frame changes by the product of the changes of the frame size, therefore the frame size of 320x240 represents a quarter of the number of pixels represented by the frame size 640x480.

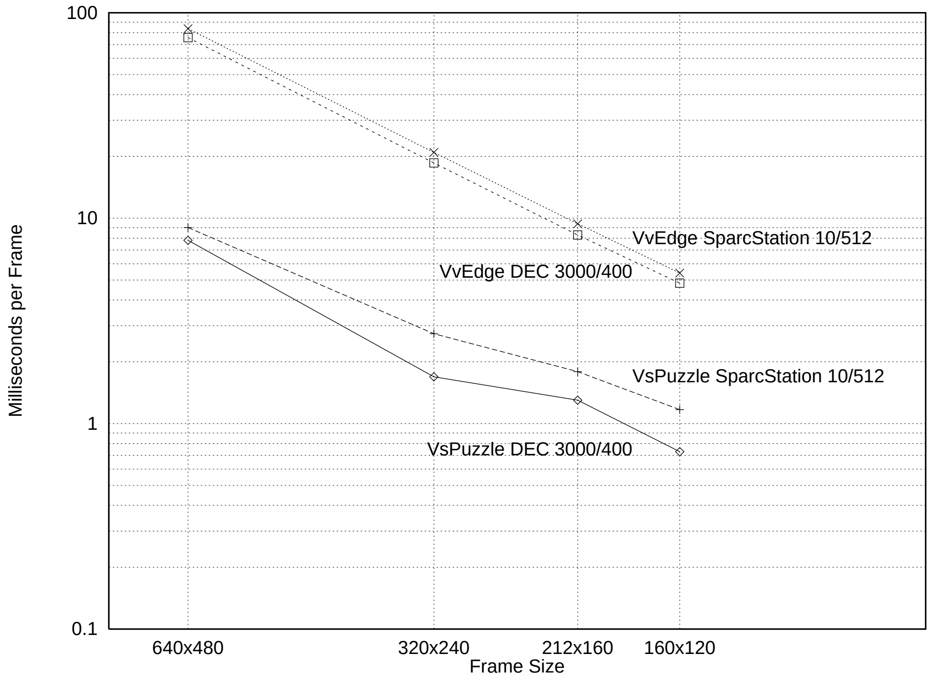


Figure 7.9: A plot of milliseconds of processing per video frame as a function of frame size.

Table 7.1 shows the filter processing times in milliseconds per frame and nanoseconds per pixel. It indicates that on both the Digital DEC 3000/400 and the Sun SparcStation 10/512, the `VsPuzzle` filter module can scramble a half-sized frame in approximately 2.5 milliseconds, and the `VvEdge` filter module can highlight edges in a half-sized frame in approximately 20 milliseconds. These times indicate that elaborate pixel-based operations can be performed efficiently on standard computer workstations.

The per-pixel performance of the `VsPuzzle` module varies with the size of its input video frame more than that of the `VvEdge` module, particularly on the Digital DEC

frame size	Digital DEC 3000/400 milliseconds per frame	Digital DEC 3000/400 nanoseconds per pixel	Sun SparcStation 10/512 milliseconds per frame	Sun SparcStation 10/512 nanoseconds per pixel
VsPuzzle				
640x480	7.81	25	9.01	29
320x240	1.69	22	2.74	36
212x160	1.30	38	1.79	52
160x120	0.73	38	1.17	61
VvEdge				
640x480	75.76	247	83.68	272
320x240	18.58	242	20.92	272
212x160	8.29	243	9.40	275
160x120	4.83	252	5.41	282

Table 7.1: Some filter processing times of representative VuSystem modules.

3000/400. Table 7.1 indicates that the VsPuzzle module gets the best per-pixel performance when run on a half-size picture on the Digital DEC 3000/400. This is because the module uses many calls to `memcpy` to re-arrange the pixels in the video frames. The performance of `memcpy` depends substantially on `memcpy` setup and cache issues. With increasing video frame size, less time per-pixel is used for setup. In the full-size case on the DEC 3000/400, the video frame size exceeds the size of the secondary cache, which causes it to have a slightly worse per-pixel performance than the half-size case does.

7.4 Timeout Precision

The VuSystem scheduler is non-preemptive, and depends on the Unix process scheduler for scheduling of time-dependent operations. To verify that this scheduling system is adequate for many perceptual-time processing tasks, I measured the exact time that `Timeout` C++ class member functions run. I instrumented the system to compare the actual time at which a `Timeout` C++ class member function is called, to the time for which `Timeout` was scheduled. If the VuSystem performs well, the difference between the actual time and the scheduled time will be very slight. I recorded this `Timeout` precision for several minute runs of the `vsdemo` program. Four runs were taken, each indicating `Timeout` precision under different system loads, created by running multiple concurrent `vsdemo` processes.

7.4.1 Experimental Setup

The `vsdemo` application is the concatenation of the `VsSource` module and the `VsSink` module. This application passes live video, audio and closed-captions from capture devices to display devices. (See page 101 for a complete discussion of the `VsSource` module, and page 102 for a complete discussion of the `VsSink` module.) Figure 7.10 shows the module configuration of the `vsdemo` program.

7.4.2 Results

Figure 7.11 shows a histogram of the percentage of `Timeout` calls, as a function of the number of milliseconds after the scheduled time that `Timeout` was called. These measurements were made on the Digital DEC 3000/400. The precision of the time-of-day clock on the Digital DEC 3000/400 is one millisecond.

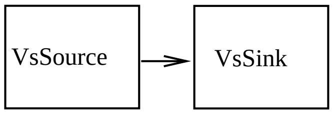


Figure 7.10: The setup to measure `Timeout` precision in the `vsdemo` VuSystem application.

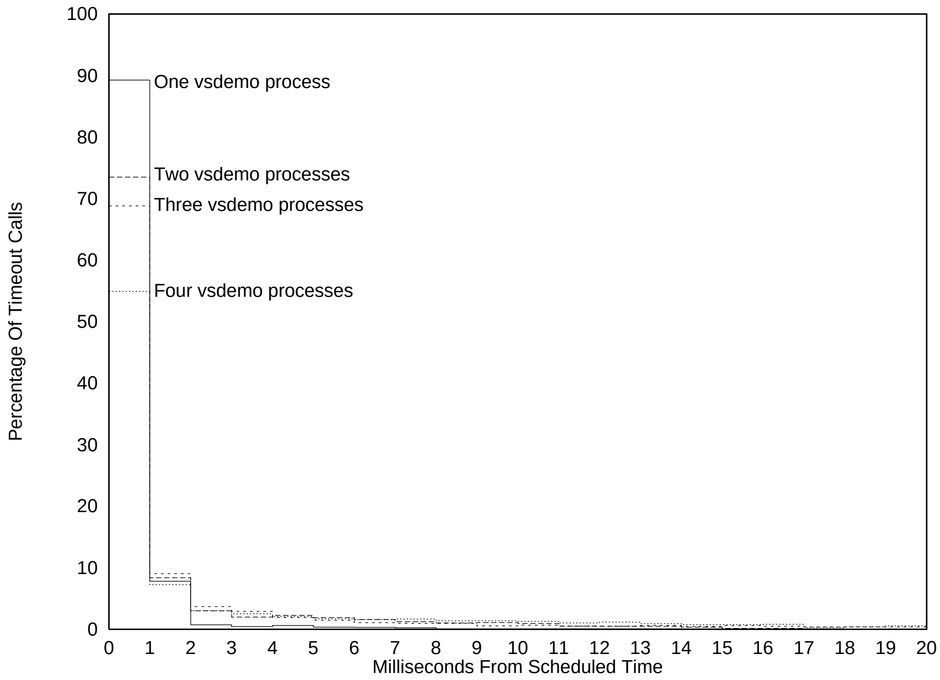


Figure 7.11: A histogram of the percentage of `Timeout` calls as a function of the number of milliseconds after the scheduled time that `Timeout` was called. (Digital DEC 3000/400).

The histogram shows that most of `Timeout` calls were made within one millisecond of the scheduled time, and can be considered "on-time". Of the calls not made within one millisecond, the vast majority were made within a few milliseconds. The histogram also shows that scheduler precision also gracefully degrades with increased system load.

7.5 Total System Throughput

I measured the the total system throughput of two VuSystem programs based on the two representative filter modules measured in Section 7.3. I measured the maximum frame rate for the `vspuzzle` application and for the `vvdemo` application.

7.5.1 Experimental Setup

The `vspuzzle` VuSystem application is the application built around `VsPuzzle` filter module. It is used as an example in Chapter 4. It scrambles a video frame to form a video puzzle. The `vvdemo` application demonstrates several features of Stasior's library of vision service modules [8]. It was used to run the `VvEdge` filter module. Figure 7.12 shows the in-band modules used in both programs.

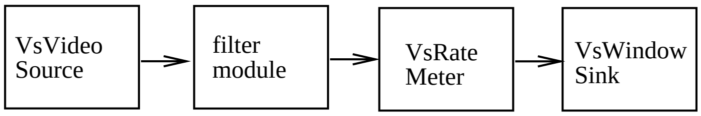


Figure 7.12: The system throughput experimental setup.

In both programs, the `VsVidboardSource` (page 119) captured a sequence of live grayscale video frame payloads. In the `vspuzzle` program, these frames were processed through the `VsPuzzle` module. In the `vvdemo` program frames were processed through the `VvEdge` module. The video frame payloads then passed through a `VsRateMeter` filter module (page 144), and finally to a `VsWindowSink` (page 135) which displayed them in a window on the computer screen. For each video frame scale factor available on the vidboard, the frame rate achieved by the programs was measured and is shown in Table 7.2.

7.5.2 Results

Table 7.2 shows the measured frame rates of simple VuSystem applications using the `VsPuzzle` and `VvEdge` filter modules on the Digital DEC 3000/400. The table indicates that the `vspuzzle` application can process half-size live video at 30 frames per second, the maximum frame rate at which video can be captured with the hardware. The `vvdemo` application running the `VvEdge` filter can process 25 frames per second of half-size live video. This verifies that applications that perform nontrivial media processing can run at perceptual-time speeds on standard computer workstations.

frame size	VsPuzzle	VvEdge
640x480	12	6.67
320x240	30	25
212x160	15	15
160x120	30	30

Table 7.2: Some measured frame rates of simple VuSystem applications using the `VsPuzzle` and `VvEdge` filter modules on the Digital DEC 3000/400.

For many of the configurations shown in Table 7.2, the system throughput was constrained not by the VuSystem, but by the Vidboard [9], the video capture hardware used. For frame sizes of 640x480, 320x240, and 160x120, the Vidboard can capture video at a maximum rate of 30 frames per second. For frame size 212x160, the Vidboard can capture video at a maximum rate of 15 frames per second [1]. This means that the throughput of `VsPuzzle` test program is constrained by the Vidboard at frame sizes of 320x240, 212x160, and 160x120; and the throughput of `VvEdge` test program is constrained by the Vidboard at frame sizes 212x160 and 160x120.

[1] To reduce grayscale video from full scale to one-third scale, the Vidboard performs a pixel filtering operation, instead of the simple pixel subsampling operation used for reduction to one-half scale and one-fourth scale.

7.6 System Throughput With Audio And Captions

I measured the total system throughput of the `vsdemo` VuSystem application. The `vsdemo` VuSystem application demonstrates the capture and display capabilities of the VuSystem through the display of live video in a window, with audio and closed-captions. It is representative of VuSystem applications that pass video, audio, and closed-captioned data. Figure 7.13 shows a logical diagram of the composite in-band modules used in the programs.

7.6.1 Experimental Setup

As discussed in Section 7.4, the `vsdemo` application is the concatenation of the `VsSource` module and the `VsSink` module. These modules are *composites*. They are implemented with several *primitive* modules. (See page 101 for a complete discussion of the `VsSource` module, and page 102 for a complete discussion of the `VsSink` module.) Of all the programs described in this chapter, the `vsdemo` program appears to use the fewest primitive modules, but actually uses the most, because it uses these complex composite modules.

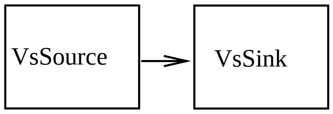


Figure 7.13: The experimental setup to measure system throughput with audio and closed-captions through the `vsdemo` VuSystem application.

For each video frame scale factor available on the vidboard, the frame rates achieved by the programs were measured and are shown in Table 7.3. Many times, the support for scaling, dithering, and closed-captions on the vidboard caused the vidboard to become the performance bottleneck for the system. Measurements were made for both color and grayscale, as well as for with and without closed-caption decoding, to show the effect of these configurations on system throughput.

7.6.2 Results

Table 7.3 shows the measured frame rates of the `vsdemo` VuSystem application on the Digital DEC 3000/400. The table indicates that the `vsdemo` application can pass half-size dithered color live video, audio, and closed-captions at 10 frames per second. Without compute-intensive closed-caption processing, half-size dithered color live video with audio can be passed at 15 frames per second. This verifies that relatively complex module configurations can pass live media and maintain synchronization at reasonable speeds on standard computer workstations without any special real-time operating system support.

7.7 Review

To verify that the overhead of the module data protocol is low, I measured the amount of time a simple transparent filter takes to process a payload. The time taken by `VsChannelSet` to process one payload was approximately 12 microseconds on the Sun SparcStation 10/512 and approximately 3 microseconds on the Digital DEC 3000/400. Given these numbers, we can conclude that the payload-passing overhead of the VuSystem was low, less than 12 microseconds on the Sun Sparcstation 10/512 and less than 3 microseconds on the Digital DEC 3000/400.

frame size	dithered color and captions	dithered color	grayscale and captions	grayscale
640x480	3.75	4.25	7	7.5
320x240	10	15	15	25
212x160	15	24	10	15
260x120	15	30	15	26

Table 7.3: Some measured frame rates of the `vsdemo` VuSystem application on the Digital DEC 3000/400.

To verify that the VuSystem run-time scheduler has low overhead, I measured the amount of time a mimimum filter with a `Work` C++ class member function takes to process a payload. The time taken by `VsFilter` to process one payload was approximately 150 microseconds on the Sun SparcStation 10/512 and approximately 115 microseconds on the Digital DEC 3000/400. Given these numbers, we can conclude that the scheduler overhead of the VuSystem for `Work` C++ class member functions was low, less than 150 microseconds on the Sun Sparcstation 10/512 and less than 115 microseconds on the Digital DEC 3000/400.

To verify that media processing modules in the VuSystem are able to perform their functions with perceptual-time granularity, I measured the amount of time two representative filter modules took to process a video frame. On both the Digital DEC 3000/400 and the Sun SparcStation 10/512, the `VsPuzzle` filter module can scramble a half-sized frame in approximately 2.5 milliseconds, and the `VvEdge` filter module can highlight edges in a half-sized frame in approximately 20 milliseconds.

To verify that this VuSystem scheduler is adequate for many perceptual-time processing tasks, I instrumented the system to compare the actual time at which a `Timeout` C++ class member function is called with the time for which the `Timeout` C++ class member function was scheduled. Most of `Timeout` calls were made within one millisecond of the scheduled time, and can be considered "on-time". Of the calls not made within one millisecond, the vast majority were made within a few milliseconds. Scheduler precision also gracefully degrades with increased system load.

I measured the total system throughput of two VuSystem programs based on two representative filter modules. The `vspuzzle` application can process half-size live video at fully 30 frames per second. The `vvdemo` application running the `VvEdge` filter can process 25 frames per second of half-size live video.

7.7.1 Summary

The overhead of the run-time component of the VuSystem is low. Representative VuSystem in-band processing modules are efficient. The VuSystem scheduler can cause operations to occur at reasonably precise times. The system has enough thoughput to support full-motion video. All these measurements verify that the VuSystem meets perceptual-time constraints sufficiently to support media-processing applications.

Chapter 8

Conclusion

With this chapter I conclude my report, beginning with a discussion of the primary contributions of this report, followed by a discussion of additional insights I have gained in the course of my research. A survey of work in progress by myself and others on the VuSystem is also included, as well as suggestions for future work.

8.1 Primary Contributions

The primary contributions of this report is the identification of *computer-participative* multimedia applications as an application class with unique requirements, and the design and implementation of a programming system that supports the development of experimental computer-participative multimedia applications.

8.1.1 Computer-Participative Multimedia Applications

I have identified a class of multimedia applications in which the computer performs tasks requiring the direct processing of multimedia data, as well as the capture, storage, retrieval, and display tasks of traditional multimedia applications. Members of the class are best called *computer-participative* multimedia applications, because in them the computer directly participates in the interpretation of the multimedia data.

Traditional multimedia toolkits are optimized for the efficient capture, storage, retrieval, and display of pre-recorded video sequences. They do not adequately support the extensible, direct *in-band* processing of multimedia data. Computer-participative multimedia applications require more support than is provided by these toolkits.

Visualization systems allow a wide variety of operations on sequences of images. They provide a library of image processing modules that can be hooked together to transform a sequence of source images stored in individual files to a sequence of resultant images. They also provide a graphical programming system that can be used to combine processing modules into programs. However, they are inadequate for multimedia applications because they do not include the synchronization support for temporally sensitive data that multimedia toolkits include.

A system that supports the development of computer-participative multimedia applications must provide a mechanism for extensible programming of multimedia data like that of visualization systems, but also must provide the temporal sensitivity and support for synchronization of multimedia systems.

8.1.2 The VuSystem

I designed and implemented the VuSystem, which supports the development of computer-participative multimedia applications. VuSystem applications are partitioned into *in-*

band code that manipulates the audio or video data, and *out-of-band* code that performs event-driven functions. My approach is unique in that it combines the computational flexibility of visualization systems with the temporal sensitivity of multimedia systems. The system is data-directed, handling dynamically-typed and self-identifying payloads. Applications are dynamically reconfigurable, even when media is flowing.

The in-band partition of the VuSystem is a reconfigurable directed graph of *modules* that logically pass time-stamped *payloads* holding media data. I designed a run-time system that supports memory management, communication, and scheduling for modules. The system is implemented for general-purpose Unix workstations running the X Window System, using no special real-time facilities. All in-band processing in the system is performed in a user-mode shell program without any special kernel modifications.

VuSystem programs have what can be called a *media-flow* architecture: code that directly processes temporally sensitive data is divided into processing *modules* arranged in data processing *pipelines*. This architecture is similar to that of some visualization systems [29, 31], but is unique in that all data is held in dynamically-typed time-stamped *payloads*, and programs can be reconfigured while they run. Timestamps allow for media synchronization, and dynamic typing and reconfiguration allows programs to change their behavior based on the data being fed into them.

A rich set of modules has been developed. Over fifty modules have been written, including filters that perform image processing and machine vision functions as well as JPEG compression and decompression [16]. A Tcl library for user-interface programming with over fifty script files has also been developed by the users. I have found it easy to reuse these modules and Tcl scripts in new applications. Performance of representative modules on DEC 3000/400 and Sun SPARCstation 10/512 workstations demonstrates that my communication protocol is efficient and practical. Applications that perform visual processing can easily do so at 15 half-resolution frames per second. This is an acceptable level of performance for today, and will improve with advances in workstation technology since the system is portable.

The out-of-band partition of the VuSystem is programmed in the Tool Command Language, or Tcl [26], an interpreted scripting language. Application code written in Tcl is responsible for creating and controlling the network of in-band media-processing modules, and controlling the graphical user-interface of the application. In-band modules are manipulated with *object commands*, and in-band events are handled with asynchronous *callbacks*.

The VuSystem is implemented on Unix workstations as a program that interprets an extended version of Tcl. This single executable image, the *application shell*, can implement many applications. All out-of-band code, including all user-interface code, is written as Tcl scripts. In-band modules are implemented as C++ classes and are linked into the shell. Application scripts written in Tcl run in the application shell, creating the modules they need. Simple applications that use the default set of in-band modules are written as Tcl scripts. Applications that require additional special-purpose in-band processing modules use *customized* application shells.

The system provides a high degree of modularity, reusability and extensibility without sacrificing performance. Developers have found it easy to build new applications with existing modules. They have found it easy to extend the system with new modules.

8.2 Additional Insights

In the process of my research I have gained several additional insights. One insight is that not only is the partitioning of in-band and out-of-band code in multimedia applications important, but that it is also important that the design of each partition allow for application extensibility. The right choice of architectural and programming language issues is important for both partitions.

8.2.1 In-Band Issues

By splitting VuSystem code in to in-band and out-of-band partitions, both partitions can be optimized differently. In-band code can be written in a low level language and can be optimized for performance, while out-of-band code can be written in a high level language and optimized for programmability, usability and extensibility.

Modules and Payloads

It is not necessarily novel to structure an in-band multimedia processing partition as a directed graph of modules. However, to have the modules pass self-identifying, dynamically typed payloads through a tight payload-passing protocol allows for the use of payloads as more than just data packages. The VuSystem uses payloads as *scheduling tokens* that automatically balance computational resources between modules through starvation and back-pressure.

Payloads also allow for *dynamic reconfigurability* of in-band processing networks. Through their descriptors, payloads themselves hold all the information about their data, whereas the connections between modules hold no descriptive information. Connections between processing modules can easily be broken and modules can be created and destroyed, all while the system is running, without losing any data or timing information. This would be much more difficult if data were passed between modules with a much simpler mechanism.

Programming Language

C++ satisfied the requirements that the programming language for in-band processing be efficient, portable, and object-oriented, but it is hardly optimal. The language scores well on efficiency and portability, but it is a poor language for implementing dynamic object oriented systems. C++ provides no mechanism for dynamically determining the class of objects, for example. I had to implement a mechanism for dynamic class determination, using virtual functions and static variables. C++ code for in-band modules and payloads also involves too much "boilerplate" code. Ideally, there should be a better language for this.

Extensibility

Some other multimedia systems partition in-band processing from out-of-band processing, but the VuSystem is unique in that it is designed to provide safe, application-level extensibility for the in-band partition as well as the out-of-band partition. This is important. Without safe application-level extensibility of the in-band partition, a programming system cannot support true computer-participative multimedia applications, because the toolkit author cannot envision all possible in-band operations that any computer-participative multimedia application might require.

Some systems manipulate media data away from the application, in an operating system kernel or server process. The motivation for their design is cheap data transfer and precise scheduling through the avoidance of standard Unix services perceived to be inefficient. By keeping all media data in a system kernel or server process, no copying is required, since all data share the same address space. Extremely precise scheduling can be maintained in an operating system kernel implementation, since the media processing code can run with very high scheduler priority and make use of hardware and software interrupts.

Unfortunately, systems built this way are difficult to extend without losing the benefits of memory protection and security. A user of such a system is either not trusted and is not allowed to add new modules to the kernel or server process, or is trusted completely and is allowed to add modules to the kernel or server process without any memory protection or security. The VuSystem does not have this problem. In the VuSystem, in-band processing is performed in a user process, so standard operating system memory-protection and security mechanisms can protect the system from errant in-band processing modules.

In the VuSystem, copying is minimized in data transfer between processes through the use of shared memory segments, and scheduling is done using standard Unix scheduling interfaces. As new standard interfaces like the POSIX real-time extensions [14] become available, even more precise scheduling capabilities will be available to Unix applications. With the VuSystem approach, a *practical* and *highly extensible* media-processing system can be implemented using standard Unix system services.

Real Time and Virtual Time

Much effort is being made in current research to design multimedia systems that associate and maintain at all times a fixed video frame rate with each desktop application. This approach is well motivated, but runs the risk of neglecting the power and elegance of the virtual model of computation that has been proven so sucessful in the past.

Classic real-time systems approximate real world time at a granularity determined by the available technology. As computers get faster, real-time constraints get tighter. In contrast, virtual-time programs operate with little or no detailed knowledge of real world time, and generally take larger inputs as computers get faster. With some assistance from the operating system, virtual-time applications *adapt* to the computational resources made available to them. This adaptability provides for *graceful degradation* of application performance, a virtue that should not be neglected in the introduction of multimedia systems.

In the VuSystem, media processing application components operate in virtual time, using closed-loop control over media source components. Temporally sensitive data can be incorporated into applications without sacrificing scalability and graceful degradation. Applications written using this approach are conveniently ported to higher performance platforms as they become available.

8.2.2 Out-of-band Issues

I found that Tcl is an excellent programming language for out-of-band control in the VuSystem. The simplicity of the language and its interpreted nature provide for the rapid prototyping of new VuSystem applications. Its trivial type system is speeding the implementation of remote-evaluation support for distributed applications. Tcl's extensibility and simple interface to C is used to a great extent in the VuSystem through object commands. Finally, the introspective features of Tcl and the VuSystem ease the development of interactive visual media programming systems.

Graphical User-Interface Toolkits and Tcl

Claims have been made that the Tk [18] graphical user-interface toolkit provides capabilities to the Tcl programmer that cannot be provided through the X Window System Toolkit [30] and the Athena widget set. The claims effectively state that even though Tcl is designed to have an efficient interface to C and Tcl-based applications can leverage off existing C libraries, existing C-based graphical user-interface libraries are not good enough for Tcl-based programs, which instead require a *totally new* graphical user-interface toolkit. These claims simply are not true. In reality, the `TclXt` and `TclXaw` libraries I developed for the VuSystem do fundamentally all of what Tk does, but use the standard Athena widget set, instead of a new, incompatible set of widgets written from scratch.

It was not hard to write `TclXt` and `TclXaw`. The X Window System Toolkit includes most of the necessary support for converting values to and from strings. After writing a simple extension to Tcl to support object-oriented programming, I wrote a Tcl command for each C library interface procedure. The result is a Tcl interface to the X Window System Toolkit and the Athena widget set.

I developed `TclXt` and `TclXaw` because at the time I started the developing VuSystem, Tcl was young and there were not many widgets written for the Tk widget set. Specifically, there was no useful text-manipulation widget for Tk. Instead of writing a text-widget for Tk or waiting for one to be written, I simple wrote a Tcl interface to the X Window System Toolkit and the Athena widget set, which already included a very useful text-manipulation widget.

Since then, Tk has gained much popularity because of its ease of programming through Tcl. Tcl/Tk fans have developed a text-manipulation widget and many other elaborate and useful Tk widgets. These widgets provide much of the capability that makes Tk so popular today. It is a waste, however, that the Tk and Xt widget sets are incompatible.

8.3 Work in Progress on the VuSystem

The VuSystem is in use at MIT and elsewhere. At MIT there are 8 developers using the environment (four graduate students and four undergraduates). The students have collectively implemented several applications for the computer-participative multimedia environment and are extending the system's scope. Work in progress includes visual processing for seamless interactive computing, a visual programming system for media computation, a distributed programming system for media applications, and a media server acessible through the World Wide Web.

8.3.1 Visual Processing For Seamless Interactive Computing

William Stasior is investigating concrete ways that computers may become more responsive and thus less visible to their human users [8]. He is developing a prototype "Computerized Office Multimedia Assistant", which will be capable of assisting its user by performing various tasks which require the analysis of time varying imagery.

Stasior is studying the role of visual processing within the framework of *seamless interactive* computing. The term interactive refers to a broadened context of interaction which includes the human, the computer, *and* the physical environment. Stasior is studying a mode of interaction which encompasses the user's complete physical environment. People work on desktops, write on blackboards, and handle documents. Stasior's ambition is to use computers to augment the interaction between people and these objects.

Such interaction will require the computer to acquire and analyze sensory input. Stasior intends to investigate, develop, and integrate various video analysis tools into a working, interactive system. He proposes to build a prototype "Computerized Office Multimedia Assistant", or COMMA. COMMA will be capable of assisting its user by performing various tasks which require the analysis of time varying imagery. For example, the user may ask his or her assistant to monitor the office, recording the identity of everybody who drops by while the user is gone. Similarly, the assistant may be asked to monitor the whiteboard and to summarize the information which is subsequently recorded.

Ultimately, the office assistant will be capable of responding to a variety of queries about the state of the office. For example, the user may wish to ask the assistant (perhaps from a phone) whether the lights are on, whether the garbage can is empty, whether the door is opened, etc. Finally, COMMA will be capable of recognizing various visual commands or gestures to facilitate the user in interacting with his or her workstation.

Stasior's applications are programmed using the VuSystem, and involve creating a library of vision service modules. Example vision service modules include a *change detector*, a *motion detector*, and a *stationary filter*. The change detector accepts two input

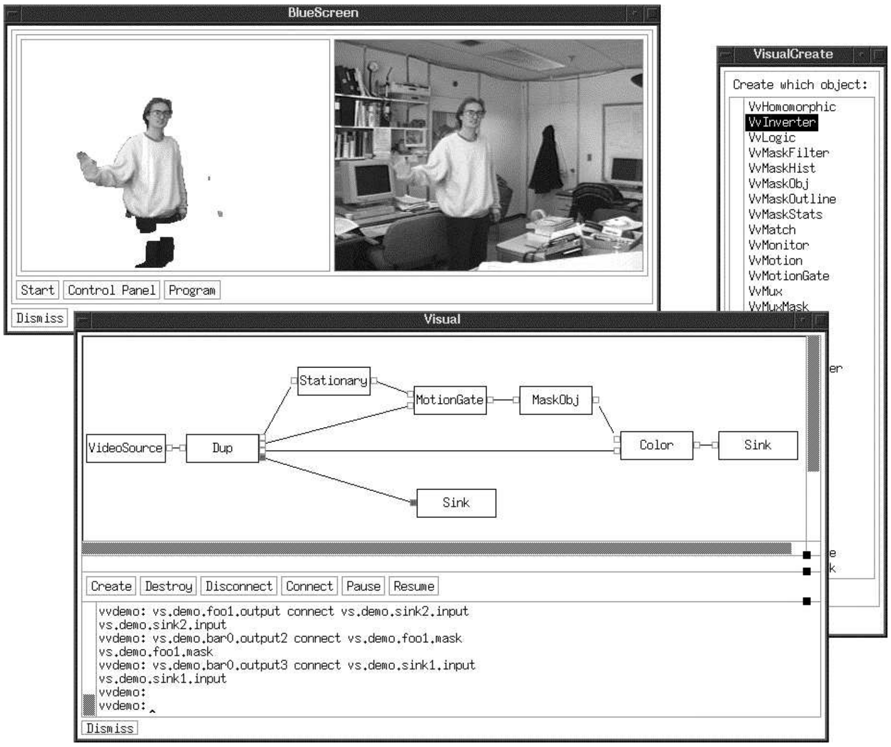


Figure 8.1: The visual programming interface to the VuSystem, manipulating some vision service modules. The program running is a smart blue-screen application, which separates the active foreground from the stationary backgound in a video sequence. The diagram is of the VuSystem modules that comprise the program.

images and outputs a binary image which shows which pixels differ on the two input images. The motion detector attempts to detect and localize motion in a video stream by outputting a true value for those pixels which correspond to moving objects in the scene. The stationary filter is effectively the converse of the motion detector.

The vision service modules would communicate with the scripting language by signaling events or callbacks to the Tcl layer. For example, a face recognition service may be implemented as a filter which calls a Tcl subroutine whenever a model face appears in the video stream. The Tcl program would, therefore, be responsible for determining how to use this information.

8.3.2 A Visual Programming System for Media Computation

David Wetherall is completing a visual programming system for application users [7]. Users interact with the system through a flow graph representation of the running program to control its media processing component. A "flow graph" perspective emphasizes the computation that occurs, rather than a "hypermedia" perspective, which may view the media in terms of a database to be navigated. Flow graph, or dataflow, representations have been used with success in prior visual languages [31].

The visual environment is suited to tasks such as customization, rapid prototyping and experimentation, as well as more general program development. It provides a programming ability (rather than a limited set of configuration options) to users, allowing them to re-program previously developed applications. By embedding it in a toolkit, consistent user programming facilities are available in all derived applications.

From the user's point of view, the visual environment consists of a number of display windows. One window shows the media flow graph representation, the primary means of user programming. Another is a customization panel, detailing the individual options that may be selected for each module of the program's computation. Further objects allow interactions with the textual programming methods of the VuSystem. Description panels show the code fragment associated with a module, and an Interpreter object evaluates commands on demand.

The visual environment demonstrates the modularity, flexibility, and extensibility of Tcl and the VuSystem. Its implementation required only a few additions to the VuSystem core; it works from within the VuSystem; and it is mostly written in Tcl. The visual environment also demonstrates the introspective capabilities of Tcl and of the VuSystem. The environment shows that VuSystem applications can examine and modify themselves while they run.

8.3.3 Distributed Programming with VuDP

Brent Phillips is developing a distributed programming system for media-based applications that provides enough support to make distributed VuSystem programs more simple and powerful [6]. Currently, all modules of a standard VuSystem program must execute in the same local environment on a single host. Applications split across the network must be realized as a set of co-operating programs, making them difficult to write. Using VuDP, a program may be constructed from modules that exist in different environments distributed across several hosts.

Under development, the VuSystem Distributed Programming (VuDP) extension will simplify the construction of applications whose processing is distributed across the network. Three advantages offered by the VuDP model are: transparent access to shared resources, the ability to divide applications across hosts, and the enabling of collaborative applications.

VuDP will support distributed programming through three mechanisms: a remote evaluation capability, an extensible set of exportable services, and a network based model for intra-application communication. The remote evaluation capability will provide a general means of creating remote modules in a suitable execution environment. The exportable services will work in an RPC-like manner to support common tasks. Finally, VuDP will support both in-band communication for media flow and out-of-band communication for control between modules at different sites.

VuDP will leverage off features of Tcl and the VuSystem that make distributed systems easy to implement. The remote evaluation component of VuDP will work by passing Tcl commands and values over reliable byte-stream TCP connections. Since all commands and values in Tcl are strings, linearization for this network transport is trivial. In addition, linearization code for in-band VuSystem payloads has already been implemented in support of the native VuSystem file format. This support will also be used for the transport of in-band data over reliable byte-stream connections. Distributed VuSystem applications will be built using this mechanism.

8.3.4 The Media Server

The Media Server has been developed to extend the reach of VuSystem applications to wide-area networks. It seamlessly integrates the *World Wide Web* with the VuSystem. By leveraging off of the network and operating system portability of the Web, and its straightforward browsing clients, The Media Server provides a publicly accessible interface to selected VuSystem applications.

The server appears to Web users as a series of pages culminating in a form that leads to video display. The Web pages act as a navigational interface to applications, using

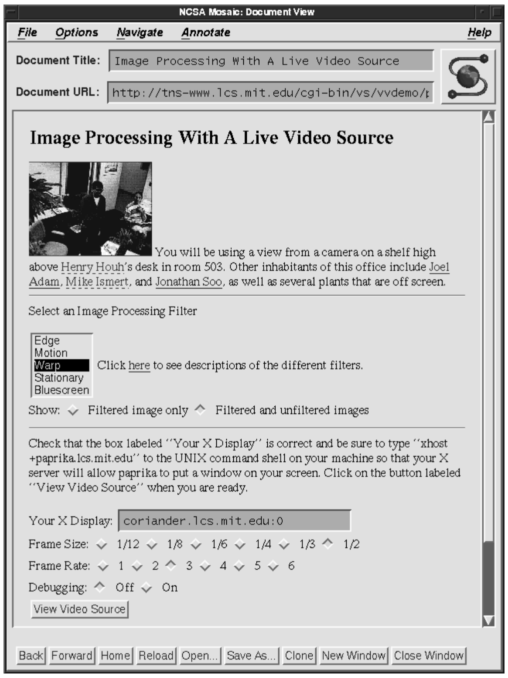


Figure 8.2: Launching an Image Processing program from the World Wide Web.

forms to select program options. Figure 8.2 shows the form used to launch a live video processing program. When a form is submitted, an application appears as though an external viewer were spawned, but without a downloading delay.

The Media Server is implemented as an HTTP server and an associated set of scripts. The scripts customize Web pages to reflect available resources and characteristics of the client. To manage network and computational load, they distribute the video processing applications across a cluster of a dozen workstations. Video files can be viewed by many clients simultaneously, but live video sources are restricted to one client at a time. The video itself is distributed using the X Window System, and audio is distributed with AudioFile [10]. This approach provides wide-area accessibility at the cost of reduced performance.

8.4 Future Work

Some directions for the future exploration of computer-participative multimedia can be explored with the VuSystem: *integration with commercial multimedia application environments*, *the application of artificial intelligence techniques*, and *the use of advanced operating system interfaces*.

Integration With Commercial Multimedia Systems

Currently, the VuSystem works as a complete stand-alone multimedia application environment. It does not interact well with other, more popular systems. It is clear that better integration of the VuSystem with commercial multimedia application environments is imperative for it to be useful to more than a small set of users. The best approach to this integration is to introduce the VuSystem techniques of in-band module programming, high level scripting, and dynamic reconfiguration into these systems.

Application Of Artificial Intelligence Techniques

The application of artificial intelligence techniques to computer-participative multimedia applications might prove fruitful. For example, machine vision and speech recognition techniques would be useful in in-band modules, and rule and production systems would be useful in the event-handling out-of-band partition of these applications.

Use Of Advanced Operating System Interfaces

The VuSystem Scheduler could be extended to make use of standard operating system interfaces that support concurrency through multi-threaded applications, such as those specified in POSIX.4 [14]. Currently, the VuSystem runs in one thread of control, without any preemption. With this framework, a `Work` procedure that runs for too long of a time, for example, may cause a `Timeout` procedure to run late. By using preemption and multiple threads of execution, the VuSystem would allow the concurrent execution of `Input`, `Output`, `Timeout` and `Work` procedures of different modules. This would have the effect of decreasing the execution granularity of the VuSystem, resulting in more temporal precision. In addition, the VuSystem could make use of emerging multiprocessor systems by having threads of execution running concurrently on multiple processors.

Any modification to the VuSystem to support multiple threads should be done with minimum impact on the programming interface. One approach might be to define each module as a critical section. The VuSystem would automatically prevent multiple threads from concurrently running code in the same module, but would allow concurrent processing between modules. Such an approach would probably have deadlock problems, but that might be resolvable.

8.5 Towards Intelligent Multimedia Environments

Multimedia systems have tended to follow one of two models: *document* based multimedia or *data-flow* based multimedia. Each of these models has its own strengths and weaknesses.

Systems based on the document model result in document processing applications. The user of these systems either authors or views multimedia documents. These systems are particularly good for using multimedia as a form of communication across time. With them, the multimedia author prepares a document to be viewed by a reader in the future. A multimedia cd-rom encyclopedia is an example of a document-based multimedia system.

Systems based on the data-flow model result in media channeling applications. The user of these systems processes large quantities of temporally sensitive data. These systems are particularly good for communication across space. With them, individuals using computers can manipulate live media. A video conferencing program is an example of a data-flow based multimedia application.

At first glance, the VuSystem fits into the data-flow of multimedia. VuSystem programs consist of a network media processing modules controlled by an application script. But the VuSystem does more. Modules in VuSystem applications perform intelligent processing of media data, and can signal the scripting level that significant events have been detected. The scripting level can reconfigure the network of modules as a reaction to the events. This combination can result in intelligent applications that can react to sensory input. I believe that the VuSystem represents the start of a new wave of *intelligent* multimedia systems.

Appendix A

Predefined Modules In The VuSystem

The modules described here were developed by several people. Many modules, including the `VsVidboardSource`, `VsPuzzle`, `VsRateMeter`, `VsJpegC`, `VsJpegD`, `VsResizeBy`, `VsResizeTo`, `VsColor24to8`, and `VsColor8to24` modules, were developed by David Wetherall. The `VsBlockShift`, `VsFade`, and `VsWipe` modules were developed by David Bacher. The `VsMpegSource` module was developed by Abhimanyu Warikoo using the Berkeley MPEG distribution. The `VsQtimeSource` and `VsQtimeSink` were developed by Peter Gloor and Ethan Mirsky using Apple Computer's Quicktime code for Unix. The `VsCCCC` and `VsCCCD` modules were developed by Andrej Duda and Ron Weiss.

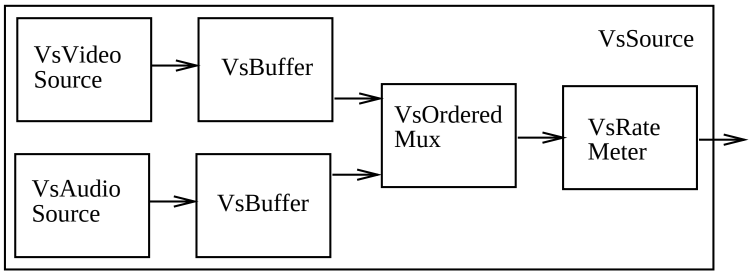


Figure A.1: The VsSource module when not reading from a file.

A.1 The VsSource Module

The VsSource Module implements a generic media source by creating appropriate primitive and composite modules based on parameters supplied to its creation command. The `-source` parameter to the VsSource module creation command specifies which primitive module to create:

- If a `-source` parameter is specified, a VsFileSource module (page 106) is created, and the value of the `-source` parameter is used as the pathname.
- If a `-source` parameter is not specified, a VsVideoSource (page 103) and a VsAudioSource (page 104) module is created. Figure A.1 shows a block diagram of this configuration.

When VsSource creates the VsVideoSource and VsAudioSource modules, it also creates other modules:

- VsBuffer modules (page 136) are created immediately downstream of the VsVideoSource and VsAudioSource modules, to slightly decouple the timing of each.
- A VsOrderedMux module (page 151) is created downstream of the VsBuffer modules, to multiplex the two input sequences to a single output sequence.
- A VsRateMeter module (page 144) is created downstream of the VsDeMux module, to measure the rate of payloads through the Source module.

Using the VsSource module creation command in scripts is preferred to other primitive and composite source module creation commands, because it provides more flexibility. An application script that uses the VsSource module creation command to create a source module can work with whatever video and audio capture interfaces are available on the local machine, working even with files.

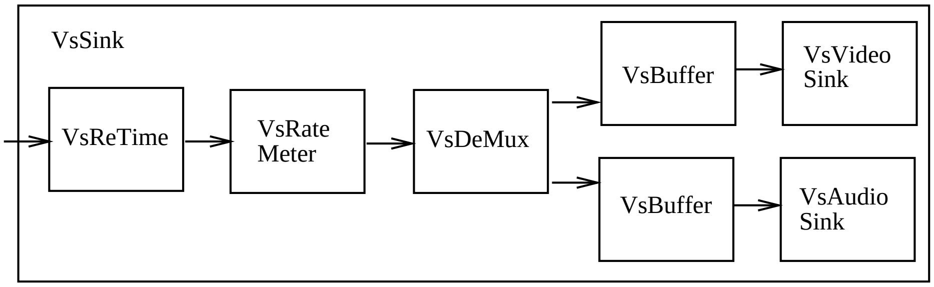


Figure A.2: The VsSink module when not writing to a file.

A.2 The VsSink Module

The VsSink Module implements a generic media sink by creating appropriate primitive and composite modules based on parameters supplied to its creation command. The `-sink` parameter to the VsSink module creation command specifies which primitive modules to create:

- If a `-sink` parameter is specified, a VsFileSink module (page 106) is created, and the value of the `-sink` parameter is used as the pathname.
- If a `-sink` parameter is not specified, a VsVideoSink (page 105) and a VsAudioSink (page 105) module is created. Figure A.2 shows a block diagram of this configuration.

When VsSink creates the VsVideoSink and VsAudioSink modules, it also creates other modules:

- VsBuffer modules (page 136) are created immediately upstream of the VsVideoSink and VsAudioSink modules, to slightly decouple the timing of each.
- A VsDeMux module (page 151) is created upstream of the VsBuffer modules, to demultiplex a single input sequence into two output sequences.

- A VsRateMeter module (page 144) is created upstream of the VsDeMux module, to measure the rate of payloads through the Sink module.
- A VsReTime module (page 145) is created upstream of the VsRateMeter module to synchronize the timestamps of payloads received by the Sink module.

Using the VsSink module creation command in scripts is preferred to other primitive and composite sink module creation commands, because it provides more flexibility. An application script that uses the VsSink command to create a sink module can work with whatever video and audio capture interfaces are available on the local machine, working even with files.

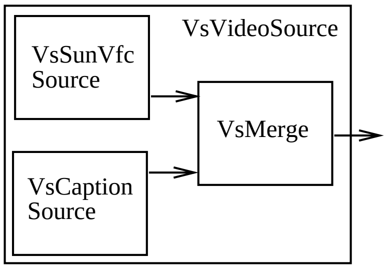


Figure A.3: The VsVideoSource module reading from a VideoPix card and a closed-caption decoder.

A.3 The VsVideoSource Module

The VsVideoSource Module implements a generic video source by creating appropriate primitive modules based on parameters supplied to its creation command. The `-videoSource` parameter to the VsVideoSource module creation command specifies which primitive modules to create:

:rtvc$N$ specifies the VsSunVideoSource module (page 116) with pathname `/dev/rtvc`N.

:vfc$N$ specifies the VsSunVfcSource module (page 114) with pathname `/dev/vfc`N.

:vidboard$N$ specifies the VsVidboardSource module (page 119) and vidboard$N$.

:xvideo specifies the VsXvideoSource module (page 126).

:test specifies the VsTestVideoSource module (page 118).

:null specifies the VsNullSource module (page 112).

anything else specifies the VsFileSource module (page 106) and pathname.

Using the VsVideoSource module creation command in scripts is preferred to primitive video source module creation commands, because it provides more flexibility. An application script that uses the VsVideoSource command to create a video source module can work with whatever video capture interface is available on the local machine, working even with files.

All the usual parameters accepted by the primitive video source modules are also accepted by the VsVideoSource module creation command. For example, the `-frameRate` parameter accepted by all the video capture modules is accepted by VsVideoSource, and passed on to the video capture module creation procedure. In addition, each parameter can be prefixed with `videoSource` to make it more specific. For example, the `-frameRate` parameter can also be supplied as `-videoSourceFrameRate`. This is handy for parameters with generic names, like `-port`. The name `-videoSourcePort` is more specific.

The VsVideoSource module also provides support for the capture of *closed-captions* as well as video. The capture of closed-captions is requested with the `-captions` *(Boolean)* parameter. Captions are captured directly with the VsVidboardSource module. For the other source modules, a VsCaptionSource (page 109) and a VsMerge (page 153) module is automatically created by VsVideoSource, and the caption payload sequence is merged with the video source payload sequence. Figure A.3 shows a block diagram of this configuration. The `-pathname` parameter to the VsCaptionSource module can be specified as `-videoSourceCaptionPathname`.

A.4 The VsAudioSource Module

The VsAudioSource Module implements a generic audio source by creating an appropriate primitive module based on parameters supplied to its creation command. The `-audioSource` parameter to the VsAudioSource module creation command specifies which primitive module to create:

:af or :audiofile specifies the VsAudioFileSource module (page 107).

:sun specifies the VsSunAudioSource module (page 113).

:dec specifies the VsDecAudioSource module (page 109).

:null specifies the VsNullSource module (page 112).

anything else specifies the VsFileSource module (page 106) and pathname.

Using the VsAudioSource module creation command in scripts is preferred to primitive audio source module creation commands, because it provides more flexibility. An application script that uses the VsAudioSource command to create an audio source module can work with whatever audio capture interface is available on the local machine, working even with files.

All the usual parameters accepted by the primitive audio source modules are also accepted by the VsAudioSource module creation command. For example, the `-gain` parameter accepted by all the audio capture modules is accepted by VsAudioSource, and passed on to the audio capture module creation procedure. In addition, each parameter can be prefixed with `audioSource` to make it more specific. For example, the `-gain` parameter can also be supplied as `-audioSourceGain`. This is handy for parameters with generic names, like `-port`. The name `-audioSourcePort` is more specific.

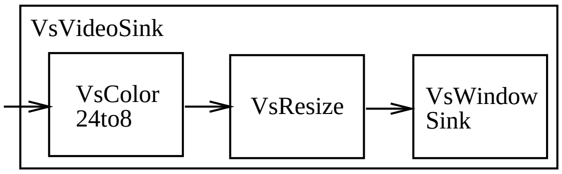


Figure A.4: The VsVideoSink module writing to an 8-bit deep window with absolute resizing.

A.5 The VsVideoSink Module

The VsVideoSink Module implements a generic video sink by creating appropriate primitive modules based on parameters supplied to its creation command. The `-videoSink` parameter to the VsVideoSink creation command specifies which primitive modules to create:

:window specifies the VsWindowSink module (page 135). Figure A.4 shows a block diagram of this configuration.

:null specifies the VsNullSink module (page 134).

anything else specifies the VsFileSink module (page 106) and pathname.

When creating a VsWindowSink module, the VsVideoSink module automatically creates a *color video frame depth conversion* module to match the depth of the window on which video is to be displayed:

- If the window has a depth of 8, VsVideoSink creates a VsColor24to8 module (page 138) and connects it upstream from the VsWindowSink module.
- If the window has a depth of 24, VsVideoSink creates a VsColor8to24 module (page 138) and connects it upstream from the VsWindowSink module.

When creating a VsWindowSink module, the VsVideoSink module also automatically creates a *color video frame resizing* module based on the value of the `-videoSinkResize` or `-resize` parameter:

- If the parameter is `absolute`, VsVideoSink creates a VsResize module (page 147) and connects it upstream from the VsWindowSink module.
- If the parameter is `relative`, VsVideoSink creates a VsScale module (page 148) and connects it upstream from the VsWindowSink module.

Using the VsVideoSink module creation command in scripts is preferred to primitive video sink module creation commands, because it provides more flexibility. An application script that uses the VsVideoSink command to create a video sink module can work with whatever video capture interface is available on the local machine, working even with files.

All the usual parameters accepted by the primitive video sink modules are also accepted by the VsVideoSink module creation command. In addition, each parameter can be prefixed with `videoSink` to make it more specific.

A.6 The VsAudioSink Module

The VsAudioSink Module implements a generic audio sink by creating an appropriate primitive module based on parameters supplied to its creation command. The `-audioSink` parameter to the VsAudioSink creation command specifies which primitive module to create:

:af or :audiofile specifies the VsAudioFileSink module (page 129).

:sun specifies the VsSunAudioSink module (page 134).

:dec specifies the VsDecAudioSink module (page 131).

:null specifies the VsNullSink module (page 134).

anything else specifies the VsFileSink module (page 106) and pathname.

Using the VsAudioSink module creation command in scripts is preferred to primitive audio sink module creation commands, because it provides more flexibility. An application script that uses the VsAudioSink command to create an audio sink module can work with whatever audio capture interface is available on the local machine, working even with files.

All the usual parameters accepted by the primitive audio sink modules are also accepted by the VsAudioSink module creation command. For example, the `-gain` parameter accepted by all the audio playback modules is accepted by VsAudioSink, and passed on to the audio capture module creation procedure. In addition, each parameter can be prefixed with `audioSink` to make it more specific. For example, the `-gain` parameter can also be supplied as `-audioSinkGain`. This is handy for parameters with generic names, like `-port`. The name `-audioSinkPort` is more specific.

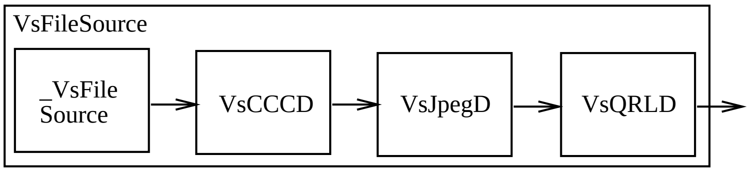


Figure A.5: The VsFileSource module with all decompression.

A.7 The VsFileSource Module

The VsFileSource Module implements a generic file source by creating a _VsFileSource primitive module (page 111) and appropriate decompression modules based on parameters supplied to its creation command. The `-fileSourceCompress` or `-compress` parameter to the VsFileSource creation command specifies which decompression modules to create upstream from the _VsFileSource module:

ccc specifies the VsCCCD module (page 137).

jpeg specifies the VsJpegD module (page 140).

qrl specifies the VsQRLD module (page 144).

all specifies the VsCCCD, the VsJpegD, and the VsQRLD modules in series. In this configuration, each module decompresses payloads that it is designed to handle, and passes transparently everything else. Figure A.5 shows a block diagram of this configuration.

none specifies no modules.

Using the VsFileSource module creation command in scripts is preferred to the _VsFileSource module creation command, because it provides more flexibility. An application script that uses the VsFileSource module creation command to create a file source module can easily select a compression technique.

All the usual parameters accepted by the primitive file source modules are also accepted by the VsFileSource module creation command. In addition, each parameter can be prefixed with `fileSource` to make it more specific.

A.8 The VsFileSink Module

The VsFileSink Module implements a generic file sink by creating a _VsFileSink primitive module (page 133) and appropriate compression modules based on parameters supplied to its creation command. The `-fileSinkCompress` or `-compress` parameter to

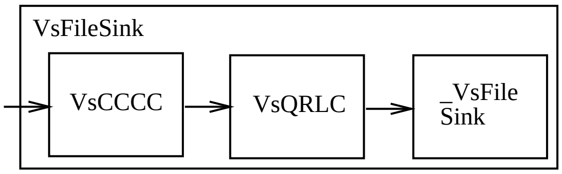


Figure A.6: The VsFileSink module with qrl+ccc compression.

the VsFileSink module creation command specifies which compression modules to create upstream from the _VsFileSink module:

ccc specifies the VsCCCC module (page 136).

jpeg specifies the VsJpegC module (page 139).

qrl specifies the VsQRLC module (page 143).

qrl+ccc specifies the VsQRLC and VsCCCC modules in series. In this configuration, the VcQRLC module compresses black and white video frames, and the VsCCCC module compresses color video frames. Figure A.6 shows a block diagram of this configuration.

none specifies no modules.

Using the VsFileSink module creation command in scripts is preferred to the _VsFileSink module creation command, because it provides more flexibility. An application script that uses the VsFileSink module creation command to create a file sink module can easily select a compression technique.

All the usual parameters accepted by the primitive file sink modules are also accepted by the VsFileSink module creation command. In addition, each parameter can be prefixed with `fileSink` to make it more specific. For example, the `-quality` compression parameter can also be supplied as `-fileSinkQuality`.

A.9 Primitive Source Modules

Source modules have no input ports and only one output port. The output port is always named `output`. Source modules are usually media capture device interface modules.

A.9.1 The VsAudioFileSource Module

The VsAudioFileSource module provides an audio source interface to the AudioFile [10] protocol. It is based on the VsEntity module.

The `server` VsAudioFileSource Subcommand

```
<vsAudioFileSource> server [<server>]
  ==> <server>
```

The `server` VsAudioFileSource subcommand provides access to the AudioFile server specifier for a VsAudioFileSource module. It takes:

server *(String)* A new AudioFile server specification.

It returns:

server *(String)* The current AudioFile server specification.

The `port` VsAudioFileSource Subcommand

```
<vsAudioFileSource> port [<port>]
  ==> <port>
```

The `port` VsAudioFileSource subcommand provides access to the audio input port specifier for a VsAudioFileSource module. It takes:

port *(Integer)* A new audio input port.

It returns:

port *(Integer)* The current audio input port.

The `numPorts` VsAudioFileSource Subcommand

```
<vsAudioFileSource> numPorts
  ==> <numPorts>
```

The `numPorts` VsAudioFileSource subcommand returns the number of input ports on an audio device. It returns:

numPorts *(Integer)* The number of input ports on the audio device.

The `gain` VsAudioFileSource Subcommand

```
<vsAudioFileSource> gain [<gain>]
  ==> <gain>
```

The `gain` VsAudioFileSource subcommand provides access to the gain setting for a VsAudioFileSource module. It takes:

gain *(Integer)* A new gain.

It returns:

gain *(Integer)* The current gain.

The `maxGain` VsAudioFileSource Subcommand

```
<vsAudioFileSource> maxGain
  ==> <maxGain>
```

The `maxGain` VsAudioFileSource subcommand returns the maximum gain value for the audio device. It returns:

maxGain *(Integer)* The maximum gain value for the audio device.

The `minGain` VsAudioFileSource Subcommand

```
<vsAudioFileSource> minGain
  ==> <minGain>
```

The `minGain` VsAudioFileSource subcommand returns the minimum gain value for the audio device. It returns:

minGain *(Integer)* The minimum gain value for the audio device.

A.9.2 The VsCaptionSource Module

The VsCaptionSource module provides a caption source interface to a closed-caption decoder that connects to the serial line of a workststion. It is based on the VsEntity module.

The `pathname` VsCaptionSource Subcommand

```
<vsCaptionSource> pathname [<pathname>]
  ==> <pathname>
```

The `pathname` VsCaptionSource subcommand provides access to the serial line device node pathname for a VsCaptionSource module. It takes:

> **pathname** *(Pathname)* A new pathname.

It returns:

> **pathname** *(Pathname)* The current pathname.

A.9.3 The VsDecAudioSource Module

The VsDecAudioSource module provides an audio source interface to the DecAudio protocol. It is based on the VsEntity module.

The `audioserver` VsDecAudioSource Subcommand

```
<vsDecAudioSource> audioserver [<audioserver>]
  ==> <audioserver>
```

The `audioserver` VsDecAudioSource subcommand provides access to the DecAudio server specifier for a VsDecAudioSource module. It takes:

> **audioserver** *(String)* A new DecAudio server specifier.

It returns:

> **audioserver** *(String)* The DecAudio server specifier.

The `port` VsDecAudioSource Subcommand

```
<vsDecAudioSource> port [<port>]
  ==> <port>
```

The `port` VsDecAudioSource subcommand provides access to the input port for the audio device. It takes:

> **port** *(Integer)* A new input port.

It returns:

> **port** *(Integer)* The current input port.

The `gain` VsDecAudioSource Subcommand

```
<vsDecAudioSource> gain [<gain>]
  ==> <gain>
```

The `gain` VsDecAudioSource subcommand provides access to the gain of the audio source. It takes:

gain *(Integer)* A new gain.

It returns:

gain *(Integer)* The current gain.

A.9.4 The VsExternalSource Module

The VsExternalSource module provides a video source interface to image sequences stored in separate image files. It handles a variety of image formats. It is based on the VsEntity module.

The VsExternalSource module indicates through its callback that it has reached end-of-file on its input. The callback command string is evaluated with the following keyword parameter appended:

-sourceEnd *(Boolean)* The module has reached end-of-file on its input.

The `type` VsExternalSource Subcommand

```
<vsExternalSource> type [<type>]
  ==> <type>
```

The `type` VsExternalSource subcommand provides access to the input type specifier for a VsExternalSource module. It takes:

type *(jpeg, pnm, or none)* A new input type.

It returns:

type *(jpeg, pnm, or none)* The current input type.

The `channel` VsExternalSource Subcommand

```
<vsExternalSource> channel [<channel>]
  ==> <channel>
```

The `channel` VsExternalSource subcommand provides access to the channel specifier for a VsExternalSource module. It takes:

channel *(Integer)* A new channel.

It returns:

channel *(Integer)* The current channel specifier.

The `pathname` VsExternalSource Subcommand

```
<vsExternalSource> pathname [<pathname>]
  ==> <pathname>
```

The `pathname` VsExternalSource subcommand provides access to the pathname parameter for a VsExternalSource module. It takes:

pathname *(Pathname)* A new pathname.

It returns:

pathname *(Pathname)* The current pathname.

The `nextFile` VsExternalSource Subcommand

```
<vsExternalSource> nextFile [<nextFile>]
  ==> <nextFile>
```

The `nextFile` VsExternalSource subcommand provides access to the file name generating command string for a VsExternalSource module. It takes:

nextFile *(Command String)* A new file name generating command string.

It returns:

nextFile *(Command String)* The current file name generating command string.

The `frameRate` VsExternalSource Subcommand

```
<vsExternalSource> frameRate [<frameRate>]
  ==> <frameRate>
```

The `frameRate` VsExternalSource subcommand provides access to the frame rate for a VsExternalSource module, in frames-per-second. It takes:

frameRate *(Double)* A new frame rate.

It returns:

frameRate *(Double)* The current frame rate.

A.9.5 The _VsFileSource Module

The _VsFileSource module provides a media source interface to files in the native VuSystem file format. It can start and end at any point in a file, and can read files backwards as well as forwards. It is based on the VsByteStream module.

The `pathname` _VsFileSource Subcommand

```
<_vsFileSource> pathname [<pathname>]
  ==> <pathname>
```

The `pathname` _VsFileSource subcommand provides access to the pathname parameter for a _VsFileSource module. It takes:

pathname *(Pathname)* A new pathname.

It returns:

pathname *(Pathname)* The current pathname.

The `reverse` _VsFileSource Subcommand

```
<_vsFileSource> reverse [<reverse>]
  ==> <reverse>
```

The `reverse` _VsFileSource subcommand provides access to the reverse parameter for a _VsFileSource module. If the reverse parameter is true, the file is read in reverse direction. It takes:

reverse *(Boolean)* A new reverse.

It returns:

reverse *(Boolean)* The current reverse.

A.9.6 The VsMpegSource Module

The VsMpegSource module provides a video source interface to files in the MPEG [21] file format. It is based on the VsEntity module.

The VsMpegSource module indicates through its callback that it has reached end-of-file on its input. The callback command string is evaluated with the following keyword parameter appended:

-sourceEnd *(Boolean)* The module has reached end-of-file on its input.

The `pathname` VsMpegSource Subcommand

```
<vsMpegSource> pathname [<pathname>]
  ==> <pathname>
```

The `pathname` VsMpegSource subcommand provides access to the pathname parameter for a VsMpegSource module. It takes:

pathname *(Pathname)* A new pathname.

It returns:

pathname *(Pathname)* The current pathname.

A.9.7 The VsNullSource Module

The VsNullSource module provides a null source. It is based on the VsEntity module.

A.9.8 The VsQtimeSource Module

The VsQtimeSource module provides a video source interface to files in the Apple Quicktime [12] file format. It is based on the VsEntity module.

The `pathname` VsQtimeSource Subcommand

```
<vsQtimeSource> pathname [<pathname>]
  ==> <pathname>
```

The `pathname` VsQtimeSource subcommand provides access to the pathname parameter for a VsQtimeSource module. It takes:

pathname *(Pathname)* A new pathname.

It returns:

pathname *(Pathname)* The current pathname.

A.9.9 The VsSunAudioSource Module

The VsSunAudioSource module provides an audio source interface to Sun audio hardware. It is based on the VsEntity module.

The `pathname` VsSunAudioSource Subcommand

```
<vsSunAudioSource> pathname [<pathname>]
  ==> <pathname>
```

The `pathname` VsSunAudioSource subcommand provides access to the audio device node pathname parameter for a VsSunAudioSource module. It takes:

pathname *(Pathname)* A new audio device node pathname.

It returns:

pathname *(Pathname)* The current audio device node pathname.

The `port` VsSunAudioSource Subcommand

```
<vsSunAudioSource> port [<port>]
  ==> <port>
```

The `port` VsSunAudioSource subcommand provides access to the input port specifier for the audio device. It takes:

port *(microphone, 1, 2, 3, or 4)* A new input port specifier.

It returns:

port *(1, 2, 3, or 4)* The current input port specifier.

The `gain` VsSunAudioSource Subcommand

```
<vsSunAudioSource> gain [<gain>]
  ==> <gain>
```

The `gain` VsSunAudioSource subcommand provides access to the gain for the audio device. It takes:

gain *(Integer)* A new gain.

It returns:

gain *(Integer)* The current gain.

The `monitorGain` VsSunAudioSource Subcommand

```
<vsSunAudioSource> monitorGain [<monitorGain>]
  ==> <monitorGain>
```

The `monitorGain` VsSunAudioSource subcommand provides access to the monitor gain parameter for the audio device. It takes:

monitorGain *(Integer)* A new monitor gain.

It returns:

monitorGain *(Integer)* The current monitor gain.

A.9.10 The VsSunVfcSource Module

The VsSunVfcSource module provides a video source interface to the Sun VideoPix video capture hardware. It is based on the VsEntity module.

The `pathname` VsSunVfcSource Subcommand

```
<vsSunVfcSource> pathname [<pathname>]
  ==> <pathname>
```

The `pathname` VsSunVfcSource subcommand provides access to the VideoPix device node pathname for the VideoPix. It takes:

> **pathname** *(Pathname)* A new VideoPix device node pathname.

It returns:

> **pathname** *(Pathname)* The current VideoPix device node pathname.

The `port` VsSunVfcSource Subcommand

```
<vsSunVfcSource> port [<port>]
  ==> <port>
```

The `port` VsSunVfcSource subcommand provides access to the input port for the VideoPix. It takes:

> **port** *(svideo, 1, 2, or 3)* A new input port.

It returns:

> **port** *(1, 2, or 3)* The current input port.

The `std` VsSunVfcSource Subcommand

```
<vsSunVfcSource> std [<std>]
  ==> <std>
```

The `std` VsSunVfcSource subcommand provides access to the video standard for the VideoPix. It takes:

> **std** *(auto, ntsc, or pal)* A new video standard.

It returns:

> **std** *(auto, ntsc, pal)* The current video standard.

The `color` VsSunVfcSource Subcommand

```
<vsSunVfcSource> color [<color>]
  ==> <color>
```

The `color` VsSunVfcSource subcommand provides access to the color switch for the VideoPix. If true, color video is captured. If false, black-and-white video is captured. It takes:

> **color** *(Boolean)* A new color switch value.

It returns:

> **color** *(Boolean)* The current color switch valie.

The `hue` VsSunVfcSource Subcommand

```
<vsSunVfcSource> hue [<hue>]
  ==> <hue>
```

The `hue` VsSunVfcSource subcommand provides access to the hue adjustment for the VideoPix, which ranges from above -180 to 180. It takes:

hue *(Integer)* A new hue.

It returns:

hue *(Integer)* The current hue.

The `scale` VsSunVfcSource Subcommand

```
<vsSunVfcSource> scale [<scale>]
  ==> <scale>
```

The `scale` VsSunVfcSource subcommand provides access to the scale parameter for the VideoPix. The scale parameter specifies the size of the video frames generated: 1 means full size, 2 means half size, and 4 means quarter size. It takes:

scale *(1, 2, or 4)* A new scale.

It returns:

scale *(1, 2, or 4)* The current scale.

The `depth` VsSunVfcSource Subcommand

```
<vsSunVfcSource> depth [<depth>]
  ==> <depth>
```

The `depth` VsSunVfcSource subcommand provides access to the depth parameter for the VideoPix, which specifies the depth of the color video frames captured. It takes:

depth *(8 or 24)* A new depth.

It returns:

depth *(8 or 24)* The current depth.

The `frameRate` VsSunVfcSource Subcommand

```
<vsSunVfcSource> frameRate [<frameRate>]
  ==> <frameRate>
```

The `frameRate` VsSunVfcSource subcommand provides access to the frame rate parameter for the VideoPix, in frames-per-second. It takes:

frameRate *(Integer)* A new frame rate.

It returns:

frameRate *(Integer)* The current frame rate.

The `byteOrder` VsSunVfcSource Subcommand

```
<vsSunVfcSource> byteOrder [<byteOrder>]
  ==> <byteOrder>
```

The `byteOrder` VsSunVfcSource subcommand provides access to the byte order parameter for the VideoPix. It specifies the byte order to use for captured video frames. It takes:

byteOrder *(msbFirst or lsbFirst)* A new byte order.

It returns:

byteOrder *(msbFirst or lsbFirst)* The current byte order.

The `encoding` VsSunVfcSource Subcommand

```
<vsSunVfcSource> encoding [<encoding>]
  ==> <encoding>
```

The `encoding` VsSunVfcSource subcommand provides access to the color pixel encoding used for 24-bit color frames captured with the VideoPix. It takes:

encoding *(bgr or rgb)* A new encoding.

It returns:

encoding *(bgr or rgb)* The current encoding.

A.9.11 The VsSunVideoSource Module

The VsSunVideoSource module provides a video source interface to the SunVideo capture hardware through Sun's XIL library. It is based on the VsEntity module.

The `pathname` VsSunVideoSource Subcommand

```
<vsSunVideoSource> pathname [<pathname>]
  ==> <pathname>
```

The `pathname` VsSunVideoSource subcommand provides access to the SunVideo device node pathname parameter for the SunVideo capture card. It takes:

pathname *(Pathname)* A new SunVideo device node pathname.

It returns:

pathname *(Pathname)* The current SunVideo device node pathname.

The `port` VsSunVideoSource Subcommand

```
<vsSunVideoSource> port [<port>]
  ==> <port>
```

The `port` VsSunVideoSource subcommand provides access to the input port specifier for the SunVideo capture card. It takes:

port *(svideo, 0, 1 or 2)* A new input port.

It returns:

port *(0, 1, or 2)* The current input port.

The `color` VsSunVideoSource Subcommand

```
<vsSunVideoSource> color [<color>]
  ==> <color>
```

The `color` VsSunVideoSource subcommand provides access to the color switch for the SunVideo capture card. If true, video is captured in color. If false, video is captured in black-and-white. It takes:

color *(Boolean)* A new color switch value.

It returns:

color *(Boolean)* The current color switch value.

The `scale` VsSunVideoSource Subcommand

```
<vsSunVideoSource> scale [<scale>]
  ==> <scale>
```

The `scale` VsSunVideoSource subcommand provides access to the scale parameter for the SunVideo capture card. The scale parameter specifies the size of the video frames generated: 1 means full size, 2 means half size, and so forth. It takes:

scale *(Integer)* A new scale.

It returns:

scale *(Integer)* The current scale.

The `depth` VsSunVideoSource Subcommand

```
<vsSunVideoSource> depth [<depth>]
  ==> <depth>
```

The `depth` VsSunVideoSource subcommand provides access to the depth parameter for the SunVideo capture card, which specifies the depth of the color video frames captured. It takes:

depth *(8 or 24)* A new depth.

It returns:

depth *(8 or 24)* The current depth.

The `frameRate` VsSunVideoSource Subcommand

```
<vsSunVideoSource> frameRate [<frameRate>]
  ==> <frameRate>
```

The `frameRate` VsSunVideoSource subcommand provides access to the frame rate parameter for the SunVideo capture card, in frames-per-second. It takes:

frameRate *(Integer)* A new frame rate.

It returns:

frameRate *(Integer)* The current frame rate.

A.9.12 The VsTestVideoSource Module

The VsTestVideoSource module provides a test video source that generates video frames of various sizes and depths. It is based on the VsEntity module.

The `scale` VsTestVideoSource Subcommand

```
<vsTestVideoSource> scale [<scale>]
  ==> <scale>
```

The `scale` VsTestVideoSource subcommand provides access to the scale parameter for a VsTestVideoSource module. The scale parameter specifies the size of the video frames generated: 1 means full size, 2 means half size, and so forth. It takes:

scale *(1, 2, 3, or 4)* A new scale.

It returns:

scale *(1, 2, 3, or 4)* The current scale.

The `depth` VsTestVideoSource Subcommand

```
<vsTestVideoSource> depth [<depth>]
  ==> <depth>
```

The `depth` VsTestVideoSource subcommand provides access to the depth parameter for a VsTestVideoSource module, which specifies the depth of the color video frames generated. It takes:

depth *(8 or 24)* A new depth.

It returns:

depth *(8 or 24)* The current depth.

The `cycle` VsTestVideoSource Subcommand

```
<vsTestVideoSource> cycle [<cycle>]
  ==> <cycle>
```

The `cycle` VsTestVideoSource subcommand provides access to the cycle length parameter for a VsTestVideoSource module, which specifies the size in megabytes of the block of memory to cycle though when generating test frames. It is useful for ensuring generated video frames are not in the cache. It takes:

cycle *(Double)* A new cycle length.

It returns:

cycle *(Double)* The current cycle length.

The `byteOrder` VsTestVideoSource Subcommand

```
<vsTestVideoSource> byteOrder [<byteOrder>]
  ==> <byteOrder>
```

The `byteOrder` VsTestVideoSource subcommand provides access to the byte order parameter for a VsTestVideoSource module, which specifies the byte order to use for generated video frames. It takes:

byteOrder *(lsbFirst or msbFirst)* A new byte order.

It returns:

byteOrder *(lsbFirst or msbFirst)* The current byte order.

The `timeStep` VsTestVideoSource Subcommand

```
<vsTestVideoSource> timeStep [<timeStep>]
  ==> <timeStep>
```

The `timeStep` VsTestVideoSource subcommand provides access to the time step parameter for a VsTestVideoSource module, which specifies the time in seconds between generated video frames. It takes:

timeStep *(Double)* A new time step.

It returns:

timeStep *(Double)* The current time step.

The `timeBase` VsTestVideoSource Subcommand

```
<vsTestVideoSource> timeBase [<timeBase>]
  ==> <timeBase>
```

The `timeBase` VsTestVideoSource subcommand provides access to the time base parameter for a VsTestVideoSource module. It takes:

timeBase *(real or virtual)* A new time base.

It returns:

timeBase *(real or virtual)* The current time base.

A.9.13 The VsVidboardSource Module

The VsVidboardSource module provides a video source interface to the Vidboard [9], a ATM-based video capture device. It is basd on the VsEntity module.

The `feOff` VsVidboardSource Subcommand

```
<vsVidboardSource> feOff
```

The `feOff` VsVidboardSource subcommand sends a front-end off command cell to the vidboard. It is only used for diagnostic purposes.

The `feInit` VsVidboardSource Subcommand

```
<vsVidboardSource> feInit
```

The `feInit` VsVidboardSource subcommand sends a front-end init command cell to the vidboard. It is only used for diagnostic purposes.

The `softInit` VsVidboardSource Subcommand

```
<vsVidboardSource> softInit
```

The `softInit` VsVidboardSource subcommand initializes everything but the front-end on the vidboard. It is only used for diagnostic purposes.

The `getFrame` VsVidboardSource Subcommand

```
<vsVidboardSource> getFrame
```

The `getFrame` VsVidboardSource subcommand sends a frame capture command cell to the vidboard. It is only used for diagnostic purposes.

The `getCaption` VsVidboardSource Subcommand

```
<vsVidboardSource> getCaption
```

The `getCaption` VsVidboardSource subcommand sends a caption capture command cell to the vidboard. It is only used for diagnostic purposes.

The `std` VsVidboardSource Subcommand

```
<vsVidboardSource> std [<std>]
  ==> <std>
```

The `std` VsVidboardSource subcommand provides access to the video standard parameter for the vidboard. It takes:

> **std** *(auto, ntsc, or pal)* A new video standard.

It returns:

> **std** *(auto, ntsc, or pal)* The current video standard.

The `port` VsVidboardSource Subcommand

```
<vsVidboardSource> port [<port>]
  ==> <port>
```

The `port` VsVidboardSource subcommand provides access to the input port specifier for the vidboard. It takes:

> **port** *(0, 1, or 2)* A new input port.

It returns:

> **port** *(0, 1, or 2)* The current input port.

The `colorSpace` VsVidboardSource Subcommand

```
<vsVidboardSource> colorSpace [<colorSpace>]
  ==> <colorSpace>
```

The `colorSpace` VsVidboardSource subcommand provides access to the color space parameter for the vidboard. It takes:

> **colorSpace** *(auto, rgb, or yuv)* A new color space.

It returns:

> **colorSpace** *(auto, rgb, or yuv)* The current color space.

The `packType` VsVidboardSource Subcommand

```
<vsVidboardSource> packType [<packType>]
  ==> <packType>
```

The `packType` VsVidboardSource subcommand provides access to the pack type parameter for the vidboard. It takes:

> **packType** *(auto, pixel, or frame)* A new pack type.

It returns:

> **packType** *(auto, pixel, or frame)* The current pack type.

The `color` VsVidboardSource Subcommand

```
<vsVidboardSource> color [<color>]
  ==> <color>
```

The `color` VsVidboardSource subcommand provides access to the color switch for the vidboard. If true, video is captured in color. If false, video is captured in black-and-white. It takes:

> **color** *(Boolean)* A new color switch value.

It returns:

> **color** *(Boolean)* The current color switch value.

The `depth` VsVidboardSource Subcommand

```
<vsVidboardSource> depth [<depth>]
  ==> <depth>
```

The `depth` VsVidboardSource subcommand provides access to the depth parameter for the vidboard, which specifies the depth of the color video frames captured. It takes:

> **depth** *(8 or 24)* A new depth.

It returns:

> **depth** *(8 or 24)* The current depth.

The `dither` VsVidboardSource Subcommand

```
<vsVidboardSource> dither [<dither>]
  ==> <dither>
```

The `dither` VsVidboardSource subcommand provides access to the dither switch for the vidboard. If true, 8-bit color frames are dithered. If false, they are not dithered. It takes:

dither *(Boolean)* A new dither switch value.

It returns:

dither *(Boolean)* The current dither switch value.

The `captions` VsVidboardSource Subcommand

```
<vsVidboardSource> captions [<captions>]
  ==> <captions>
```

The `captions` VsVidboardSource subcommand provides access to the captions channel for the vidboard. If non-zero, it specified which closed-caption channel to capture. If zero, captions are ignored. It takes:

captions *(0, 1, or 2)* A new captions channel.

It returns:

captions *(0, 1, or2)* The current captions channel.

The `hue` VsVidboardSource Subcommand

```
<vsVidboardSource> hue [<hue>]
  ==> <hue>
```

The `hue` VsVidboardSource subcommand provides access to the hue adjustment for the vidboard, which ranges from above -180 to 180. It takes:

hue *(Integer)* A new hue.

It returns:

hue *(Integer)* The current hue.

The `scale` VsVidboardSource Subcommand

```
<vsVidboardSource> scale [<scale>]
  ==> <scale>
```

The `scale` VsVidboardSource subcommand provides access to the scale parameter for the vidboard. The scale parameter specifies the size of the video frames captured: 1 means full size, 2 means half size, and so forth. It takes:

scale *(1, 2, 3, or 4)* A new scale.

It returns:

scale *(1, 2, 3, or 4)* The current scale.

The `byteOrder` VsVidboardSource Subcommand

```
<vsVidboardSource> byteOrder [<byteOrder>]
  ==> <byteOrder>
```

The `byteOrder` VsVidboardSource subcommand provides access to the byte order parameter for the vidboard, which specifies the byte order to use for captured video frames. It takes:

> **byteOrder** *(lsbFirst or msbFirst)* A new byte order.

It returns:

> **byteOrder** *(lsbFirst or msbFirst)* The current byte order.

The `encoding` VsVidboardSource Subcommand

```
<vsVidboardSource> encoding [<encoding>]
  ==> <encoding>
```

The `encoding` VsVidboardSource subcommand provides access to color pixel encoding to use for 24-bit color frames captured with the vidboard. It takes:

> **encoding** *(bgr or rgb)* A new color pixel encoding.

It returns:

> **encoding** *(bgr or rgb)* The current color pixel encoding.

The `frameRate` VsVidboardSource Subcommand

```
<vsVidboardSource> frameRate [<frameRate>]
  ==> <frameRate>
```

The `frameRate` VsVidboardSource subcommand provides access to the frame rate parameter for the vidboard, in frames-per-second It takes:

> **frameRate** *(Double)* A new frame rate.

It returns:

> **frameRate** *(Double)* The current frame rate.

The `vciLocalDataIn` VsVidboardSource Subcommand

```
<vsVidboardSource> vciLocalDataIn [<vciLocalDataIn>]
  ==> <vciLocalDataIn>
```

The `vciLocalDataIn` VsVidboardSource subcommand provides access to the vciLocalDataIn parameter for the vidboard, which specifies the local input vci for the data circuit. It takes:

> **vciLocalDataIn** *(Integer)* A new vci.

It returns:

> **vciLocalDataIn** *(Integer)* The current vci.

The `vciLocalControlOut` VsVidboardSource Subcommand

```
<vsVidboardSource> vciLocalControlOut [<vciLocalControlOut>]
  ==> <vciLocalControlOut>
```

The **`vciLocalControlOut`** VsVidboardSource subcommand provides access to the vciLocalControlOut parameter for the vidboard, which specifies the local output vci for the control circuit. It takes:

vciLocalControlOut *(Integer)* A new vci.

It returns:

vciLocalControlOut *(Integer)* The current vci.

The `vciLocalControlIn` VsVidboardSource Subcommand

```
<vsVidboardSource> vciLocalControlIn [<vciLocalControlIn>]
  ==> <vciLocalControlIn>
```

The **`vciLocalControlIn`** VsVidboardSource subcommand provides access to the vciLocalControlIn parameter for the vidboard, which specifies the local input vci for the control circuit. It takes:

vciLocalControlIn *(Integer)* A new vci.

It returns:

vciLocalControlIn *(Integer)* The current vci.

The `vciRemoteDataOut` VsVidboardSource Subcommand

```
<vsVidboardSource> vciRemoteDataOut [<vciRemoteDataOut>]
  ==> <vciRemoteDataOut>
```

The **`vciRemoteDataOut`** VsVidboardSource subcommand provides access to the vciRemoteDataOut parameter for the vidboard, which specifies the remote output vci for the data circuit. It takes:

vciRemoteDataOut *(Integer)* A new vci.

It returns:

vciRemoteDataOut *(Integer)* The current vci.

The `vciRemoteControlOut` VsVidboardSource Subcommand

```
<vsVidboardSource> vciRemoteControlOut [<vciRemoteControlOut>]
  ==> <vciRemoteControlOut>
```

The **`vciRemoteControlOut`** VsVidboardSource subcommand provides access to the vciRemoteControlOut parameter for the vidboard, which specifies the remote output vci for the control circuit. It takes:

vciRemoteControlOut *(Integer)* A new vci.

It returns:

vciRemoteControlOut *(Integer)* The current vci.

The `tportRemoteData` VsVidboardSource Subcommand

```
<vsVidboardSource> tportRemoteData [<tportRemoteData>]
  ==> <tportRemoteData>
```

The `tportRemoteData` VsVidboardSource subcommand provides access to the tportRemoteData parameter for the vidboard, which specifies the remote tport for the data circuit. It takes:

> **tportRemoteData** *(Integer)* A new tport.

It returns:

> **tportRemoteData** *(Integer)* The current tport.

The `tportRemoteControl` VsVidboardSource Subcommand

```
<vsVidboardSource> tportRemoteControl [<tportRemoteControl>]
  ==> <tportRemoteControl>
```

The `tportRemoteControl` VsVidboardSource subcommand provides access to the tportRemoteControl parameter for the vidboard, which specifies the remote tport to use for the control circuit. It takes:

> **tportRemoteControl** *(Integer)* A new tport.

It returns:

> **tportRemoteControl** *(Integer)* The current tport.

The `interDatagramDelay` VsVidboardSource Subcommand

```
<vsVidboardSource> interDatagramDelay [<interDatagramDelay>]
  ==> <interDatagramDelay>
```

The `interDatagramDelay` VsVidboardSource subcommand provides access to the interDatagramDelay parameter for the vidboard, which specifies the delay in microseconds between datagrams sent by the vidboard. It takes:

> **interDatagramDelay** *(auto or an Integer)* A new delay.

It returns:

> **interDatagramDelay** *(auto or an Integer)* The current delay.

The `interBurstDelay` VsVidboardSource Subcommand

```
<vsVidboardSource> interBurstDelay [<interBurstDelay>]
  ==> <interBurstDelay>
```

The `interBurstDelay` VsVidboardSource subcommand provides access to the interBurstDelay parameter for the vidboard, which specifies the delay in microseconds between cell bursts sent by the vidboard. It takes:

> **interBurstDelay** *(auto or an Integer)* A new delay.

It returns:

> **interBurstDelay** *(auto or an Integer)* The current delay.

The `cellsPerBurst` VsVidboardSource Subcommand

```
<vsVidboardSource> cellsPerBurst [<cellsPerBurst>]
  ==> <cellsPerBurst>
```

The `cellsPerBurst` VsVidboardSource subcommand provides access to the cellsPerBurst parameter for the vidboard, which specifies the number of cells in a cell burst. It takes:

> **cellsPerBurst** *(auto or an Integer)* A new number.

It returns:

> **cellsPerBurst** *(auto or an Integer)* The current number.

The `linesPerDatagram` VsVidboardSource Subcommand

```
<vsVidboardSource> linesPerDatagram [<linesPerDatagram>]
  ==> <linesPerDatagram>
```

The `linesPerDatagram` VsVidboardSource subcommand provides access to the linesPerDatagram parameter for the vidboard, which specifies the number of image scan lines to include per datagram sent by the vidboard. It takes:

> **linesPerDatagram** *(auto or an Integer)* A new number.

It returns:

> **linesPerDatagram** *(auto or an Integer)* The current number.

A.9.14 The VsXVideoSource Module

The VsXVideoSource module provides a video source interface to frame buffers with image capture capabilities through the XVideo X server extension. It is based on the VsEntity module.

The `port` VsXVideoSource Subcommand

```
<vsXVideoSource> port [<port>]
  ==> <port>
```

The `port` VsXVideoSource subcommand provides access to the input port parameter for an XVideo capture card. It takes:

> **port** *(Integer)* A new input port.

It returns:

> **port** *(Integer)* The current input port.

The `numPorts` VsXVideoSource Subcommand

```
<vsXVideoSource> numPorts
  ==> <numPorts>
```

The `numPorts` VsXVideoSource subcommand returns the number of input video ports. It returns:

> **numPorts** *(Integer)* The number of input ports.

The `frameRate` VsXVideoSource Subcommand

```
<vsXVideoSource> frameRate [<frameRate>]
  ==> <frameRate>
```

The `frameRate` VsXVideoSource subcommand provides access to the frame rate parameter for an XVideo capture card, in frames-per-second. It takes:

frameRate *(Integer)* A new frame rate.

It returns:

frameRate *(Integer)* The current frame rate.

The `hue` VsXVideoSource Subcommand

```
<vsXVideoSource> hue [<hue>]
  ==> <hue>
```

The `hue` VsXVideoSource subcommand provides access to the hue adjustment for an XVideo capture card. It ranges between -1000 and 1000. It takes:

hue *(Integer)* A new hue.

It returns:

hue *(Integer)* The current hue.

The `saturation` VsXVideoSource Subcommand

```
<vsXVideoSource> saturation [<saturation>]
  ==> <saturation>
```

The `saturation` VsXVideoSource subcommand provides access to the saturation parameter for an XVideo capture card. It ranges between -1000 and 1000. It takes:

saturation *(Integer)* A new saturation.

It returns:

saturation *(Integer)* The current saturation.

The `brightness` VsXVideoSource Subcommand

```
<vsXVideoSource> brightness [<brightness>]
  ==> <brightness>
```

The `brightness` VsXVideoSource subcommand provides access to the brightness parameter for an XVideo capture card. It ranges between -1000 and 1000. It takes:

brightness *(Integer)* A new brightness.

It returns:

brightness *(Integer)* The current brightness.

The `contrast` VsXVideoSource Subcommand

```
<vsXVideoSource> contrast [<contrast>]
  ==> <contrast>
```

The `contrast` VsXVideoSource subcommand provides access to the contrast parameter for an XVideo capture card. It ranges between -1000 and 1000. It takes:

contrast *(Integer)* A new contrast.

It returns:

contrast *(Integer)* The current contrast.

The `scale` VsXVideoSource Subcommand

```
<vsXVideoSource> scale [<scale>]
  ==> <scale>
```

The `scale` VsXVideoSource subcommand provides access to the scale parameter for an XVideo capture card. The scale parameter specifies the size of the video frames captured: 1 means full size, 2 means half size, and so forth. It takes:

scale *(1, 2, or 4)* A new scale.

It returns:

scale *(1, 2, or 4)* The current scale.

The `std` VsXVideoSource Subcommand

```
<vsXVideoSource> std [<std>]
  ==> <std>
```

The `std` VsXVideoSource subcommand provides access to the video standard parameter for an XVideo capture card. It takes:

std *(String)* A new video standard.

It returns:

std *(String)* The current video standard.

The `signalType` VsXVideoSource Subcommand

```
<vsXVideoSource> signalType [<signalType>]
  ==> <signalType>
```

The `signalType` VsXVideoSource subcommand provides access to the signal type parameter for an XVideo capture card. It takes:

signalType *(String)* A new signal type.

It returns:

signalType *(String)* The current signal type.

The `widget` VsXVideoSource Subcommand

```
<vsXVideoSource> widget [<widget>]
  ==> <widget>
```

The `widget` VsXVideoSource subcommand provides access to the widget parameter for an XVideo capture card, which specifies the window in which to capture video. It takes:

> **widget** *(Command Name)* A new widget.

It returns:

> **widget** *(Command Name)* The current widget.

A.10 Primitive Sink Modules

Sink modules have one onput port and no output ports. The input port is always named `input`. Sink modules are usually media playback device interface modules.

A.10.1 The VsAudioFileSink Module

The VsAudioFileSink module provides an audio sink interface to the AudioFile [10] protocol. It is based on the VsEntity module.

The `server` VsAudioFileSink Subcommand

```
<vsAudioFileSink> server [<server>]
  ==> <server>
```

The `server` VsAudioFileSink subcommand provides access to the AudioFile server specifier for a VsAudioFileSink module. It takes:

> **server** *(String)* A new AudioFile server specifier.

It returns:

> **server** *(String)* The current AudioFile server specifier.

The `port` VsAudioFileSink Subcommand

```
<vsAudioFileSink> port [<port>]
  ==> <port>
```

The `port` VsAudioFileSink subcommand provides access to the output port parameter for a VsAudioFileSink module. It takes:

> **port** *(Integer)* A new output port bitmask.

It returns:

> **port** *(Integer)* The current output port bitmask.

The `numPorts` VsAudioFileSink Subcommand

```
<vsAudioFileSink> numPorts
  ==> <numPorts>
```

The `numPorts` VsAudioFileSink subcommand returns the number of output audio ports. It returns:

> **numPorts** *(Integer)* The number of output ports.

The `gain` VsAudioFileSink Subcommand

```
<vsAudioFileSink> gain [<gain>]
  ==> <gain>
```

The `gain` VsAudioFileSink subcommand provides access to the gain parameter for a VsAudioFileSink module. It takes:

> **gain** *(Integer)* A new gain.

It returns:

> **gain** *(Integer)* The current gain.

The `maxGain` VsAudioFileSink Subcommand

```
<vsAudioFileSink> maxGain
  ==> <maxGain>
```

The `maxGain` VsAudioFileSink subcommand returns the maximum gain value for the audio device. It returns:

> **maxGain** *(Integer)* The maximum gain value.

The `minGain` VsAudioFileSink Subcommand

```
<vsAudioFileSink> minGain
  ==> <minGain>
```

The `minGain` VsAudioFileSink subcommand returns the minimum gain value for the audio device. It returns:

> **minGain** *(Integer)* The minimum gain value.

A.10.2 The VsCaptionSink Module

The VsCaptionSink module calls its callback when it receives a caption. It ignores all other payloads. It is based on the VsEntity module.

The VsCaptionSink module indicates through its callback that it has received a `VsCaption` payload. The callback command string is evaluated with the following keyword parameter appended:

> **-caption** *(String)* The caption text.

A.10.3 The VsDecAudioSink Module

The VsDecAudioSink module provides an audio sink interface to the DecAudio protocol. It is based on the VsEntity module.

The `audioserver` VsDecAudioSink Subcommand

```
<vsDecAudioSink> audioserver [<audioserver>]
  ==> <audioserver>
```

The `audioserver` VsDecAudioSink subcommand provides access to the DecAudio server specifier parameter for a VsDecAudioSink module. It takes:

audioserver *(String)* A new DecAudio server specifier.

It returns:

audioserver *(String)* The current DecAudio server specifier.

The `port` VsDecAudioSink Subcommand

```
<vsDecAudioSink> port [<port>]
  ==> <port>
```

The `port` VsDecAudioSink subcommand provides access to the output port parameter for a VsDecAudioSink module. It takes:

port *(Integer)* A new output port bitmask.

It returns:

port *(Integer)* The current output port bitmask.

The `gain` VsDecAudioSink Subcommand

```
<vsDecAudioSink> gain [<gain>]
  ==> <gain>
```

The `gain` VsDecAudioSink subcommand provides access to the gain parameter for a VsDecAudioSink module. It takes:

gain *(Integer)* A new gain.

It returns:

gain *(Integer)* The current gain.

A.10.4 The VsExternalSink Module

The VsExternalSink module provides a video sink interface to images stored in separate files. It supports a variety of file formats, including raw, standalone, ppm, pbm, pbm-ascii, and postscript. It is based on the VsEntity module.

The VsExternalSink module indicates through its callback that it has received a `VsFinish` payload. The callback command string is evaluated with any of the following keyword parameters appended:

-sinkFinish *(Boolean)* The module has received a `VsFinish` payload but was not stopping.

-sinkStop *(Boolean)* The module has received a `VsFinish` payload and has completely stopped.

The `payload` VsExternalSink Subcommand

```
<vsExternalSink> payload [<payload>]
  ==> <payload>
```

The `payload` VsExternalSink subcommand provides access to the payload type parameter for a VsExternalSink module. It takes:

payload *(String)* A new payload type.

It returns:

payload *(String)* The current payload type.

The `channel` VsExternalSink Subcommand

```
<vsExternalSink> channel [<channel>]
  ==> <channel>
```

The `channel` VsExternalSink subcommand provides access to the channel parameter for a VsExternalSink module. It takes:

channel *(Integer)* A new channel.

It returns:

channel *(Integer)* The current channel.

The `pathname` VsExternalSink Subcommand

```
<vsExternalSink> pathname [<pathname>]
  ==> <pathname>
```

The `pathname` VsExternalSink subcommand provides access to the pathname parameter for a VsExternalSink module. It takes:

pathname *(Pathname)* A new pathname.

It returns:

pathname *(Pathname)* The current pathname.

The `nextFile` VsExternalSink Subcommand

```
<vsExternalSink> nextFile [<nextFile>]
  ==> <nextFile>
```

The `nextFile` VsExternalSink subcommand provides access to the file name generating command string for a VsExternalSink module. It takes:

nextFile *(Command String)* A new file name generating command string.

It returns:

nextFile *(Command String)* The current file name generating command string.

The `convert` VsExternalSink Subcommand

```
<vsExternalSink> convert [<convert>]
  ==> <convert>
```

The `convert` VsExternalSink subcommand provides access to the conversion specifier for a VsExternalSink module. It takes:

convert *(null, raw, standalone, ppm, pbm, pbm-ascii, or postscript)* A new conversion specifier.

It returns:

convert *(null, raw, standalone, ppm, pbm, pbm-ascii, or postscript)* The current conversion specifier.

A.10.5 The _VsFileSink Module

The _VsFileSink module provides a media sink interface to files in the native VuSystem file format. It is based on the VsByteStream module.

The `pathname` _VsFileSink Subcommand

```
<_vsFileSink> pathname [<pathname>]
  ==> <pathname>
```

The `pathname` _VsFileSink subcommand provides access to the pathname parameter for a _VsFileSink module. It takes:

pathname *(Pathname)* A new pathname.

It returns:

pathname *(Pathname)* The current pathname.

The `indexExtension` _VsFileSink Subcommand

```
<_vsFileSink> indexExtension [<indexExtension>]
  ==> <indexExtension>
```

The `indexExtension` _VsFileSink subcommand provides access to the index extension parameter for a VsFileSink module, which specifies the file name extension to use for the generated index file. It takes:

indexExtension *(String)* A new extension.

It returns:

indexExtension *(String)* The current extension.

The `payload` _VsFileSink Subcommand

```
<_vsFileSink> payload [<payload>]
  ==> <payload>
```

The `payload` _VsFileSink subcommand provides access to the payload specifier for a VsFileSink module, which specifies which payload types to write to the file. It takes:

payload *(String)* A new list of payload types.

It returns:

payload *(String)* The current list of payload types.

A.10.6 The VsNullSink Module

The VsNullSink module is a pure data sink. It is based on the VsEntity module.

The VsNullSink module indicates through its callback that it has received a finish payload while stopping. The callback command string is evaluated with any of the following keyword parameters appended:

-sinkFinish *(Boolean)* The module has received a `VsFinish` payload but was not stopping.

-sinkStop *(Boolean)* The module has received a `VsFinish` payload while stopping and has completely stopped.

A.10.7 The VsQtimeSink Module

The VsQtimeSink module provide a video sink interface to files in the Apple Quicktime [12] file format. It is based on the VsEntity module.

The VsQtimeSink module indicates through its callback that it has received a `VsFinish` payload. The callback command string is evaluated with any of the following keyword parameters appended:

-sinkFinish *(Boolean)* The module has received a `VsFinish` payload but was not stopping.

-sinkStop *(Boolean)* The module has received a `VsFinish` payload and has completely stopped.

The `pathname` VsQtimeSink Subcommand

```
<vsQtimeSink> pathname [<pathname>]
  ==> <pathname>
```

The `pathname` VsQtimeSink subcommand provides access to the pathname parameter for a VsQtimeSink module. It takes:

pathname *(Pathname)* A new pathname.

It returns:

pathname *(Pathname)* The current pathname.

A.10.8 The VsSunAudioSink Module

The VsSunAudioSink module provides an audio sink interface to Sun audio hardware. It is based on the VsEntity module.

The `pathname` VsSunAudioSink Subcommand

```
<vsSunAudioSink> pathname [<pathname>]
  ==> <pathname>
```

The `pathname` VsSunAudioSink subcommand provides access to the pathname parameter for a VsSunAudioSink module. It takes:

pathname *(Pathname)* A new pathname.

It returns:

pathname *(Pathname)* The current pathname.

The `port` VsSunAudioSink Subcommand

```
<vsSunAudioSink> port [<port>]
  ==> <port>
```

The `port` VsSunAudioSink subcommand provides access to the output port parameter for a VsSunAudioSink module. It takes:

port *(internal, external, 1, 2, 3, or 4)* A new output port.

It returns:

port *(1, 2, 3, or 4)* The current output port.

The `gain` VsSunAudioSink Subcommand

```
<vsSunAudioSink> gain [<gain>]
  ==> <gain>
```

The `gain` VsSunAudioSink subcommand provides access to the gain parameter for a VsSunAudioSink module. It takes:

gain *(Integer)* A new gain.

It returns:

gain *(Integer)* The current gain.

A.10.9 The VsWindowSink Module

The VsWindowSink module provides a video sink interface to a workstation screen through the X Window System. It is based on the VsEntity module.

The VsWindowSink module indicates through its callback that it has received a `VsFinish` payload. It also indicates through its callback that it has received a `VsCaption` payload. The callback command string is evaluated with any of the following keyword parameters appended:

-sinkFinish *(Boolean)* The module has received a `VsFinish` payload but was not stopping.

-sinkStop *(Boolean)* The module has received a `VsFinish` payload and has completely stopped.

-caption *(String)* The caption text.

The `widget` VsWindowSink Subcommand

```
<vsWindowSink> widget [<widget>]
  ==> <widget>
```

The `widget` VsWindowSink subcommand provides access to the widget parameter for a VsWindowSink module, which specifies the window in which to display video frames. It takes:

widget *(Command Name)* A new widget.

It returns:

widget *(Command Name)* The current widget.

The `grab` VsWindowSink Subcommand

```
<vsWindowSink> grab pathname
```

The `grab` VsWindowSink subcommand grabs the currently displayed video frame and puts it in a file. It takes:

> **pathname** *(Pathname)* the pathname of the file in which to put the video frame.

A.11 Primitive Filter Modules

Filter modules have one input port and one output port. The input port is always named `input`, and the output port is always named `output`. Filter modules are usually media-processing modules.

A.11.1 The VsBuffer Module

The VsBuffer module provides a buffering mechanism. The depth of the buffer is specified in time differences, instead of numbers of payloads. The VsBuffer module accepts payloads, even if its downstream module is not accepting payloads, until the range of times in the buffer exceeds the depth parameter. It is based on the VsEntity module.

The `depth` VsBuffer Subcommand

```
<vsBuffer> depth [<depth>]
  ==> <depth>
```

The `depth` VsBuffer subcommand provides access to the depth parameter for a VsBuffer module, which specifies number of seconds of time difference allowed between the timestamp on the head payload in the buffer and the timestamp on the tail payload in the buffer. It takes:

> **depth** *(Double)* A new depth.

It returns:

> **depth** *(Double)* The current depth.

A.11.2 The VsCCCC Module

The VsCCCC module does Color Cell compression. It converts each 8-bit color video frame passed to it to a compressed video frame. All other payloads are passed transparently. It is based on the VsFilter module.

The VsCCCC module indicates compression ratios through its callback. The callback command string is evaluated with the following keyword parameter appended:

> **-compressRatio** *(Float)* The compression ratio achieved.

The `reportInterval` VsCCCC Subcommand

```
<vsCCCC> reportInterval [<reportInterval>]
  ==> <reportInterval>
```

The `reportInterval` VsCCCC subcommand provides access to the report interval parameter for a VsCCCC module, which specifies the number of seconds between calls to the callback reporting compression ratios. It takes:

reportInterval *(Long)* A new report interval.

It returns:

reportInterval *(Seconds)* The current report interval.

A.11.3 The VsCCCD Module

The VsCCCD module does Color Cell decompression. It converts each color-cell compressed video frame passed to it to an 8-bit color video frame. All other payloads are passed transparently. It is based on the VsFilter module.

A.11.4 The VsChannelSelect Module

The VsChannelSelect module passes all payloads with a channel descriptor member that matches the module's channel parameter. It deletes all other payloads. It is based on the VsEntity module.

The `channel` VsChannelSelect Subcommand

```
<vsChannelSelect> channel [<channel>]
  ==> <channel>
```

The `channel` VsChannelSelect subcommand provides access to the channel parameter for a VsChannelSelect module. It takes:

channel *(Integer)* A new channel.

It returns:

channel *(Integer)* The current channel.

A.11.5 The VsChannelSet Module

The VsChannelSet module sets the channel payload descriptor member of all payloads that pass through it to the module's channel parameter. It is based on the VsEntity module.

The `channel` VsChannelSet Subcommand

```
<vsChannelSet> channel [<channel>]
  ==> <channel>
```

The `channel` VsChannelSet subcommand provides access to the channel parameter for a VsChannelSet module. It takes:

channel *(Integer)* A new channel.

It returns:

channel *(Integer)* The current channel.

A.11.6 The VsColor8to24 Module

The VsColor8to24 module converts 8-bit color video frames into 24-bit color video frames. All other payloads are passed transparently. It is based on the VsEntity module.

The `byteOrder` VsColor8to24 Subcommand

```
<vsColor8to24> byteOrder [<byteOrder>]
  ==> <byteOrder>
```

The `byteOrder` VsColor8to24 subcommand provides access to the byte order parameter for a VsColor8to24 module, which specifies the byte order for converted video frames. It takes:

byteOrder *(lsbFirst or msbFirst)* A new byte order.

It returns:

byteOrder *(lsbFirst or msbFirst)* The current byte order.

The `encoding` VsColor8to24 Subcommand

```
<vsColor8to24> encoding [<encoding>]
  ==> <encoding>
```

The `encoding` VsColor8to24 subcommand provides access to the encoding parameter for a VsColor8to24 module, which specifies the pixel encoding of converted video frames. It takes:

encoding *(bgr or rgb)* A new pixel encoding.

It returns:

encoding *(bgr or rgb)* The current pixel encoding.

A.11.7 The VsColor24to8 Module

The VsColor24to8 module converts 24-bit color video frames into 8-bit color video frames. All other payloads are passed transparently. It is based on the VsEntity module.

A.11.8 The VsExercise Module

The VsExercise module does data walking exercises on video frames. It is used to measure performance of the VuSystem. It is based on the VsFilter module.

The `cycles` VsExercise Subcommand

```
<vsExercise> cycles [<cycles>]
  ==> <cycles>
```

The `cycles` VsExercise subcommand provides access to the cycles parameter for a VsExercise module. It takes:

cycles *(Integer)* A new number of cycles.

It returns:

cycles *(Integer)* The current number of cycles.

The `transferUnit` VsExercise Subcommand

```
<vsExercise> transferUnit [<transferUnit>]
  ==> <transferUnit>
```

The `transferUnit` VsExercise subcommand provides access to the transfer unit parameter for a VsExercise module. It takes:

transferUnit *(8, 16, 32, or 64)* A new transfer unit.

It returns:

transferUnit *(8, 16, 32, or 64)* The current transfer unit.

The `mode` VsExercise Subcommand

```
<vsExercise> mode [<mode>]
  ==> <mode>
```

The `mode` VsExercise subcommand provides access to the mode parameter for a VsExercise module. It takes:

mode *(read, write, readWrite, or copy)* A new mode.

It returns:

mode *(read, write, readWrite, or copy)* The current mode.

The `microOp` VsExercise Subcommand

```
<vsExercise> microOp [<microOp>]
  ==> <microOp>
```

The `microOp` VsExercise subcommand provides access to the micro op parameter for a VsExercise module. It takes:

microOp *(Integer)* A new micro operation.

It returns:

microOp *(Integer)* The current micro operation.

A.11.9 The VsJpegC Module

The VsJpegC module performs JPEG compression. It converts all color video frames into JPEG frames. All other payloads are passed transparently. It is based on the VsFilter module.

The VsJpegC module indicates compression ratios through its callback. The callback command string is evaluated with the following keyword parameter appended:

-compressRatio *(Float)* The compression ratio achieved.

The `quality` VsJpegC Subcommand

```
<vsJpegC> quality [<quality>]
  ==> <quality>
```

The `quality` VsJpegC subcommand provides access to the output quality parameter for a VsJpegC module, which ranges between 0 and 100. It takes:

quality *(Integer)* A new output quality.

It returns:

quality *(Integer)* The current output quality.

A.11.10 The VsJpegD Module

The VsJpegD module performs JPEG decompression. It converts all JPEG frames into color video frames. All other payloads are passed transparently. It is based on the VsFilter module.

The `byteOrder` VsJpegD Subcommand

```
<vsJpegD> byteOrder [<byteOrder>]
  ==> <byteOrder>
```

The `byteOrder` VsJpegD subcommand provides access to the byte order parameter for a VsJpegD module, which specifies the byte order for decompressed video frames. It takes:

byteOrder *(lsbFirst or msbFirst)* A new byte order.

It returns:

byteOrder *(lsbFirst or msbFirst)* The current byte order.

The `encoding` VsJpegD Subcommand

```
<vsJpegD> encoding [<encoding>]
  ==> <encoding>
```

The `encoding` VsJpegD subcommand provides access to the encoding parameter for a VsJpegD module, which specifies the pixel encoding to use for decompressed 24-bit color frames. It takes:

encoding *(0, 1, 2, or 3)* A new pixel encoding.

It returns:

encoding *(0, 1, 2, or 3)* The current pixel encoding.

A.11.11 The VsPayloadDetect Module

The VsPayloadDetect module detects payloads of a certain type. It calls its callback when a payload of a specified type passes through it. It passes all payloads transparently. It is based on the VsFilter module.

The VsPayloadDetect module indicates through its callback that it has detected a payload of the right type. The callback command string is evaluated with the following keyword parameter appended:

-detect *(String)* The type of the payload that has been detected.

The `payload` VsPayloadDetect Subcommand

```
<vsPayloadDetect> payload [<payload>]
  ==> <payload>
```

The `payload` VsPayloadDetect subcommand provides access to the payload type parameter for a VsPayloadDetect module. It takes:

payload *(String)* A new payload type.

It returns:

payload *(String)* The current payload type.

A.11.12 The VsPayloadFilter Module

The VsPayloadFilter module only passes payloads of a certain type. All payloads not of the same type as the payload type parameter are deleted. It is based on the VsFilter module.

The `payload` VsPayloadFilter Subcommand

```
<vsPayloadFilter> payload [<payload>]
  ==> <payload>
```

The `payload` VsPayloadFilter subcommand provides access to the payload type parameter for a VsPayloadFilter module. It takes:

payload *(String)* A new payload type.

It returns:

payload *(String)* The current payload type.

A.11.13 The VsPuzzle Module

The VsPuzzle module scrambles video frames to form a video puzzle. It can also solve the puzzle on its own. It is based on the VsFilter module.

The VsPuzzle module indicates through its callback that the puzzle has been solved. It also indicates through its callback that it has completed a permutation. The callback command string is evaluated with any of the following keyword parameters appended:

-solved *(Boolean)* The puzzle has been solved.

-permute *(Boolean)* The permutation has completed.

The `position` VsPuzzle Subcommand

```
<vsPuzzle> position [<position>]
  ==> <position>
```

The `position` VsPuzzle subcommand provides access to the position parameter for a VsPuzzle module, which specifies where the "hole" in the puzzle is. It takes:

position *(Pair of Integers)* A new position.

It returns:

position *(Pair of Integers)* The current position.

The `dimension` VsPuzzle Subcommand

```
<vsPuzzle> dimension [<dimension>]
  ==> <dimension>
```

The `dimension` VsPuzzle subcommand provides access to the dimension parameter for a VsPuzzle module, which specifies the number of rows and columns in the puzzle. It takes:

dimension *(Integer)* A new dimension.

It returns:

dimension *(Integer)* The current dimension.

The `scramble` VsPuzzle Subcommand

```
<vsPuzzle> scramble
```

The `scramble` VsPuzzle subcommand scrambles the puzzle.

The `solve` VsPuzzle Subcommand

```
<vsPuzzle> solve [<solve>]
  ==> <solve>
```

The `solve` VsPuzzle subcommand provides access to the solve switch for a VsPuzzle module. If true, the puzzle will solve itself, one move at a time. It takes:

solve *(Boolean)* A new solve switch value.

It returns:

solve *(Boolean)* The current solve switch value.

The `permute` VsPuzzle Subcommand

```
<vsPuzzle> permute <fromPos> <toPos>
```

The `permute` VsPuzzle subcommand permutes the puzzle, moving a piece from one position to another. It takes:

fromPos *(Pair of Integers)* A starting position.

It returns:

toPos *(Pair of Integers)* An ending position.

The `lock` VsPuzzle Subcommand

```
<vsPuzzle> lock <position> [<lock>]
  ==> <lock>
```

The `lock` VsPuzzle subcommand locks or unlocks a piece at a position. It takes:

position *(Pair of Integers)* A position.

lock *(Boolean)* Lock value.

It returns:

lock *(Boolean)* The lock value for the position.

The `timeStep` VsPuzzle Subcommand

```
<vsPuzzle> timeStep [<timeStep>]
  ==> <timeStep>
```

The `timeStep` VsPuzzle subcommand provides access to the time step parameter for a VsPuzzle module, which specifies the number of seconds between moves when solving automatically. It takes:

timeStep *(Float)* A new time step.

It returns:

timeStep *(Float)* The current time step.

A.11.14 The VsQRLC Module

The VsQRLC module performs a quantized-run-length compression. It converts black-and-white video frames into compressed QRL frames. All other payloads are passed transparently. It is based on the VsFilter module.

The VsQRLC module indicates compression ratios through its callback. The callback command string is evaluated with the following keyword parameter appended:

-compressRatio *(Float)* The compression ratio achieved.

The `quality` VsQRLC Subcommand

```
<vsQRLC> quality [<quality>]
  ==> <quality>
```

The `quality` VsQRLC subcommand provides access to the output quality parameter for a VsQRLC module, which ranges between 0 and 100. It takes:

quality *(Integer)* A new output quality.

It returns:

quality *(Integer)* The current output quality.

The `reportInterval` VsQRLC Subcommand

```
<vsQRLC> reportInterval [<reportInterval>]
  ==> <reportInterval>
```

The `reportInterval` VsQRLC subcommand provides access to the report interval parameter for a VsQRLC module, which specifies the number of seconds between callbacks reporting compression ratios. It takes:

reportInterval *(Integer)* A new report interval.

It returns:

reportInterval *(Integer)* The current report interval.

The `keyFrameInterval` VsQRLC Subcommand

```
<vsQRLC> keyFrameInterval [<keyFrameInterval>]
  ==> <keyFrameInterval>
```

The **`keyFrameInterval`** VsQRLC subcommand provides access to the key frame interval parameter for a VsQRLC module, which specifies the number of microseconds between key frames. It takes:

> **keyFrameInterval** *(Integer)* A new key frame interval.

It returns:

> **keyFrameInterval** *(Integer)* The current key frame interval.

A.11.15 The VsQRLD Module

The VsQRLD module performs a quantized-run-length decompression. It converts compressed QRL frames into black-and-white video frames. All other payloads are passed transparently. It is based on the VsFilter module.

A.11.16 The VsRateMeter Module

The VsRateMeter module measures the rate of payloads passing though it. It measures the rate in virtual time, using the timestamps in the payload descriptors. It passes all payloads transparently. It is based on the VsEntity module.

The VsRateMeter module indicates payload rates through its callback. The callback command string is evaluated with the following keyword parameter appended:

> **-rate** *(Float)* The payload rate achieved, in payloads-per-second.

The `history` VsRateMeter Subcommand

```
<vsRateMeter> history [<history>]
  ==> <history>
```

The **`history`** VsRateMeter subcommand provides access to the history parameter for a VsRateMeter module, which specifies the number of microseconds of history to keep when computing payload rates. It takes:

> **history** *(Integer)* A new history.

It returns:

> **history** *(Integer)* The current history.

The `report` VsRateMeter Subcommand

```
<vsRateMeter> report [<report>]
  ==> <report>
```

The **`report`** VsRateMeter subcommand provides access to the report parameter for a VsRateMeter module, which specifies the number of microseconds between callbacks reporting payload rates. It takes:

> **report** *(Integer)* A new report interval.

It returns:

> **report** *(Integer)* The current report interval.

The `payload` VsRateMeter Subcommand

```
<vsRateMeter> payload [<payload>]
  ==> <payload>
```

The `payload` VsRateMeter subcommand provides access to the payload parameter for a VsRateMeter module, which specifies the payload type to measure the rate of. It takes:

> **payload** *(String)* A new payload type.

It returns:

> **payload** *(String)* The current payload type.

The `channel` VsRateMeter Subcommand

```
<vsRateMeter> channel [<channel>]
  ==> <channel>
```

The `channel` VsRateMeter subcommand provides access to the channel parameter for a VsRateMeter module, which specifes which channel to measure the rate of. It takes:

> **channel** *(Integer)* A new channel.

It returns:

> **channel** *(Integer)* The current channel.

The `rate` VsRateMeter Subcommand

```
<vsRateMeter> rate
  ==> <rate>
```

The `rate` VsRateMeter subcommand returns the payload rate. It returns:

> **rate** *(Float)* The payload rate.

A.11.17 The VsReTime Module

The VsReTime module is used for media synchronization. It modifies the `StartingTime` payload descriptor member of payloads that pass through it. It is based on the VsEntity module.

By adding a fixed offset to every timestamp, the filter allows the playback of media data at a time later than it was captured. The offset corresponds to the time difference between the time of day of the start of playback of a sequence, and the time of day of the start of capture of the sequence.

How It Works

Consider a sequence of media payloads, perhaps a sequence of video frames interleaved with corresponding audio fragments. When each video frame and audio fragment was captured by the VuSystem, the exact time of day of capture was recorded by the capturing *source* module in the `StartingTime` payload descriptor member for the payload. When the sequence is played back, the respective playback *sink* module presents the video frame or audio fragment at the time indicated by the `StartingTime` payload descriptor member. It is the job of the `VsReTime` module to change the `StartingTime` payload descriptor members so that the payload sequence can be played back correctly. It does so by keeping invariant the relative payload timestamps within the sequence:

$$T_{N,Playback} - T_{0,Playback} = T_{N,Capture} - T_{0,Capture} \tag{A.1}$$

The `VsReTime` module assigns the current time, plus a small offset to allow for some buffering before playback, to the `StartingTime` payload descriptor member for the first payload of the sequence:

$$T_{0,Playback} = CurrentTime + delay \tag{A.2}$$

For the rest of the payloads, `VsReTime` uses the time assigned to the first payload in the sequence and Equation A.1:

$$T_{N,Playback} = T_{N,Capture} - T_{0,Capture} + T_{0,Playback} \tag{A.3}$$

The `VsRetime` filter can also cause the playback of stored media data at a speed different than it was captured. To do this, the filter subtracts the timestamp of the first payload of the sequence, scales the result, and then adds the time of day of the start of playback of the sequence. Equation A.1 can be rewritten to include a scale factor:

$$T_{N,Playback} - T_{0,Playback} = \frac{T_{N,Capture} - T_{0,Capture}}{speed} \tag{A.4}$$

Equation A.3 can also be rewritten to include this scale factor:

$$T_{N,Playback} = \frac{T_{N,Capture} - T_{0,Capture}}{speed} + T_{0,Playback} \tag{A.5}$$

The `delay` VsReTime Subcommand

```
<vsReTime> delay [<delay>]
  ==> <delay>
```

The `delay` VsReTime subcommand provides access to the delay parameter for a VsReTime module, which specifies the delay, in seconds from the current time, that new payload timestamps should start with. It takes:

delay *(Double)* A new delay.

It returns:

delay *(Double)* The current delay.

The speed VsReTime Subcommand

```
<vsReTime> speed [<speed>]
  ==> <speed>
```

The `speed` VsReTime subcommand provides access to the speed parameter for a VsReTime module, which specifies the speed at which new payload timestamps should progress: 1 means normal time, 2 means double speed, .5 means half speed, and so forth. It takes:

speed *(Double)* A new speed.

It returns:

speed *(Double)* The current speed.

A.11.18 The VsResize Module

The VsResize module changes the size of video frames that pass through it. All other payloads are passed transparently. It is based on the VsFilter module.

The scale VsResize Subcommand

```
<vsResize> scale [<scale>]
  ==> <scale>
```

The `scale` VsResize subcommand provides access to the scale parameter for a VsResize module, which specifies how much to resize video frames. It takes:

scale *(Float)* A new scale.

It returns:

scale *(Float)* The current scale.

The width VsResize Subcommand

```
<vsResize> width [<width>]
  ==> <width>
```

The `width` VsResize subcommand provides access to the width parameter for a VsResize module. It takes:

width *(Integer)* A new width.

It returns:

width *(Integer)* The current width.

The height VsResize Subcommand

```
<vsResize> height [<height>]
  ==> <height>
```

The `height` VsResize subcommand provides access to the height parameter for a VsResize module. It takes:

height *(Integer)* A new height.

It returns:

height *(Integer)* The current height.

A.11.19 The VsScale Module

The VsScale module rescales video frames that are passed through it. The horizontal and vertical dimensions can be independently scaled. All other payloads are passed transparently. It is based on the VsFilter module.

The `scale` VsScale Subcommand

```
<vsScale> scale [<scale>]
  ==> <scale>
```

The **`scale`** VsScale subcommand provides access to the scale parameter for a VsScale module. It takes:

> **scale** *(Float)* A new scale.

It returns:

> **scale** *(Float)* The current scale.

The `xmag` VsScale Subcommand

```
<vsScale> xmag [<xmag>]
  ==> <xmag>
```

The **`xmag`** VsScale subcommand provides access to the xmag parameter for a VsScale module. It takes:

> **xmag** *(Float)* A new xmag.

It returns:

> **xmag** *(Float)* The current xmag.

The `ymag` VsScale Subcommand

```
<vsScale> ymag [<ymag>]
  ==> <ymag>
```

The **`ymag`** VsScale subcommand provides access to the ymag parameter for a VsScale module. It takes:

> **ymag** *(Float)* A new ymag.

It returns:

> **ymag** *(Float)* The current ymag.

A.11.20 The VsStepper Module

The VsStepper module provides precise control of payload passing. The stepper normally passes all payloads, but when it encounters a payload of the specified type, it passes it, calls its callback, and stops. It passes no more payloads until it is restarted. It is based on the VsEntity module.

The VsStepper module indicates completion of the step by calling its callback. The callback command string is evaluated with the following keyword parameter appended:

> **-stepDone** *(Boolean)* The step has completed.

The `payload` VsStepper Subcommand

```
<vsStepper> payload [<payload>]
  ==> <payload>
```

The `payload` VsStepper subcommand provides access to the payload type parameter for a VsStepper module. It takes:

> **payload** *(String)* A new payload type.

It returns:

> **payload** *(String)* The current payload type.

A.11.21 The VsByteStream Module

The VsByteStream module is the base module for all modules that pass payloads though byte streams in the native VuSystem format. It knows how to convert payloads from and to sequences of bytes. Modules built on the VsByteStream module include _VsFileSource, VsFileSink, VsTcpClient, and VsTcpServer. It is based on the VsEntity module.

The VsBytestream module indicates through its callback that it has reached end-of-file on its input. It also indicates through its callback that it has received a `VsFinish` payload. The callback command string is evaluated with any of the following keyword parameters appended:

> **-sourceEnd** *(Boolean)* The module has reached end-of-file on its input.
>
> **-sinkFinish** *(Boolean)* The module has received a `VsFinish` payload but was not stopping.
>
> **-sinkStop** *(Boolean)* The module has received a `VsFinish` payload and has completely stopped.

The `end` VsByteStream Subcommand

```
<vsByteStream> end [<end>]
  ==> <end>
```

The `end` VsByteStream subcommand provides access to the end parameter for a VsByteStream module, which sets the position in the byte stream to stop reading. It takes:

> **end** *(Integer)* A new end position.

It returns:

> **end** *(Integer)* The current end position.

The `seek` VsByteStream Subcommand

```
<vsByteStream> seek [<seek>]
  ==> <seek>
```

The `seek` VsByteStream subcommand provides access to the seek parameter for a VsByteStream module, which specifies the starting position in the byte stream. It takes:

> **seek** *(Integer)* A new seek position.

It returns:

> **seek** *(Integer)* The current seek position.

The `tell` VsByteStream Subcommand

```
<vsByteStream> tell
  => <position>
```

The **tell** VsByteStream subcommand returns the current seek position on the byte stream. It returns:

position *(Integer)* The current position.

A.11.22 The VsTcpClient Module

The VsTcpClient module provides an interface to the client side of a TCP connection. It is based on the VsByteStream module. It is classified as a filter because it has an input port and an output port. All payloads passed into its input port come out of the output port of a corresponding VsTcpServer module on the server side of the TCP connection. Similarly, all payloads that are passed into the input port of a corresponding VsTcpServer module on the server side of the TCP connection come out of the output port of this module.

The `host` VsTcpClient Subcommand

```
<vsTcpClient> host [<host>]
  ==> <host>
```

The `host` VsTcpClient subcommand provides access to the host parameter for a VsTcpClient module, which specifies the remote host for the TCP connection. It takes:

host *(String)* A new host name.

It returns:

host *(String)* The current host name.

The `port` VsTcpClient Subcommand

```
<vsTcpClient> port [<port>]
  ==> <port>
```

The `port` VsTcpClient subcommand provides access to the port parameter for a VsTcpClient module, which specifies the TCP port for the TCP connection. It takes:

port *(TCP Service Name or Integer)* A new TCP port.

It returns:

port *(Integer)* The current TCP port.

A.11.23 The VsTcpServer Module

The VsTcpServer module provides an interface to the server side of a TCP connection. It is based on the VsByteStream module. It is classified as a filter because it has an input port and an output port. All payloads passed into its input port come out of the output port of a corresponding VsTcpClient module on the client side of the TCP connection. Similarly, all payloads that are passed into the input port of a corresponding VsTcpClient module on the client side of the TCP connection come out of the output port of this module.

VsTcpServer modules are not created with creation commands. Instead, the VsTcpListener module (page 156) creates a VsTcpServer module whenever it receives a connection request.

A.12 Other Primitive Modules

Other modules are modules with more than one input port or more than one output port.

A.12.1 The VsBlockShift Module

The VsBlockShift module performs a time-based block-shift effect on video frames. All other payloads are passed transparently. It is based on the VsEffect module.

The VsBlockShift module indicates completion of the effect by calling its callback. The callback command string is evaluated with the following keyword parameter appended:

-done *(Boolean)* The effect has completed.

The `direction` VsBlockShift Subcommand

```
<vsBlockShift> direction [<direction>] ==> <direction>
```

The `direction` VsBlockShift subcommand provides access to the direction parameter for a VsBlockShift module. It takes:

direction *(corners or center)* A new direction.

It returns:

direction *(corners or center)* The current direction.

A.12.2 The VsDeMux Module

The VsDeMux module performs communications-like payload demultiplexing. It is based on the VsEntity module. It has one input port, named `input`; and any number of output ports, each named `output`N (for N from 0 to $numOutputPorts - 1$). Payload sequences that have been multiplexed by VsMux (page 154) or VsOrderedMux (page 155) are demultiplexed with VsDeMux. Multiplexing is performed by saving the input port number in the `channel` payload descriptor member of each payload.

The `numOutputPorts` VsDeMux Subcommand

```
<vsDeMux> numOutputPorts [<numOutputPorts>]
  ==> <numOutputPorts>
```

The `numOutputPorts` VsDeMux subcommand provides access to the numOutputPorts parameter for a VsDeMux module, which specifies the number of output ports the module should have. It takes:

numOutputPorts *(Integer)* A new number of output ports.

It returns:

numOutputPorts *(Integer)* The current number of output ports.

A.12.3 The VsDup Module

The VsDup module duplicates payload sequences. It is based on the VsEntity module. It has one input port, named `input`; and any number of output ports, each named `output`N (for N from 0 to $numOutputPorts - 1$). Payload sequences are duplicated by making shallow copies (page 53) that are distributed to each output port. A new payload is not accepted from an upstream module until all copies of an old payload have been accepted by all downstream modules.

The `numOutputPorts` VsDup Subcommand

```
<vsDup> numOutputPorts [<numOutputPorts>]
  ==> <numOutputPorts>
```

The `numOutputPorts` VsDup subcommand provides access to the numOutputPorts parameter for a VsDup module, which specifies the number of output ports the module should have. It takes:

numOutputPorts *(Integer)* A new number of output ports.

It returns:

numOutputPorts *(Integer)* The current number of output ports.

A.12.4 The VsEffect Module

The VsEffect module is the base module for modules that perform visual effects by generating a single output video sequence from two input video sequences. It is based on the VsEntity module. It has two input ports, named `input0` and `input1`, and one output port, named `output`. Modules based on the VsEffect module include VsBlockShift, VsFade, and VsWipe.

The `value` VsEffect Subcommand

```
<vsEffect> value [<value>]
  ==> <value>
```

The `value` VsEffect subcommand provides access to the value parameter for a VsEffect module, which ranges from 0 to 128. The value parameter corresponds to the lever on an effects generator box. It takes:

value *(Integer)* A new value.

It returns:

value *(Integer)* The current value.

The `duration` VsEffect Subcommand

```
<vsEffect> duration [<duration>]
  ==> <duration>
```

The `duration` VsEffect subcommand provides access to the duration parameter for a VsEffect module, which specifies the number of seconds an effect should take. It takes:

duration *(Integer)* A new duration.

It returns:

duration *(Integer)* The current duration.

The `startValue` VsEffect Subcommand

```
<vsEffect> startValue [<startValue>]
  ==> <startValue>
```

The `startValue` VsEffect subcommand provides access to the startValue parameter for a VsEffect module, which ranges from 0 to 128. It takes:

startValue *(Integer)* A new startValue.

It returns:

startValue *(Integer)* The current startValue.

The `endValue` VsEffect Subcommand

```
<vsEffect> endValue [<endValue>]
  ==> <endValue>
```

The `endValue` VsEffect subcommand provides access to the endValue parameter for a VsEffect module, which ranges from 0 to 128. It takes:

endValue *(Integer)* A new endValue.

It returns:

endValue *(Integer)* The current endValue.

A.12.5 The VsFade Module

The VsFade module performs a time-based fade effect on video frames. All other payloads are passed transparently. It is based on the VsEffect module.

The VsFade module indicates completion of the effect by calling its callback. The callback command string is evaluated with the following keyword parameter appended:

-done *(Boolean)* The effect has completed.

A.12.6 The VsMerge Module

The VsMerge module merges two payload sequences into one. It works like the VsOrderedMerge module (page 154), except that it does not ensure that the payload timestamps in the output sequence are always increasing. It has any number of input ports, each named `input`N (for N from 0 to $numInputPorts-1$); and one output port, named `output`. It is based on the VsEntity module.

The `numInputPorts` VsMerge Subcommand

```
<vsMerge> numInputPorts [<numInputPorts>]
  ==> <numInputPorts>
```

The `numInputPorts` VsMerge subcommand provides access to the numInputPorts parameter for a VsMerge module, which specifies the number of inputs ports the module should have. It takes:

numInputPorts *(Integer)* A new number of input ports.

It returns:

numInputPorts *(Integer)* The current number of input ports.

A.12.7 The VsMux Module

The VsMux module multiplexes many payload sequences into one. It is based on the VsEntity module. It works like the VsOrderedMux module (page 155), except that it does not ensure that the payload timestamps in the output sequence are always increasing. It has any number of input ports, each named `input`N (for N from 0 to $numInputPorts - 1$); and one output port, named `output`. Payload sequences that have been multiplexed by VsMux can be demultiplexed with VsDeMux (page 151). Multiplexing is performed by saving the input port number in the `channel` payload descriptor member of each payload.

A `VsMux` or `VsOrderedMux` module with n input ports records which input port from which payloads came, by updating the `Channel` payload descriptor so that $channel$ mod n returns which input port from which the payload came, and $channel/n$ returns the original `Channel` descriptor. A `VsDeMux` module with n output ports uses $channel$ mod n to select which output port to direct a payload, and replaces the `Channel` payload descriptor component with $channel/n$.

Since `Channel` payload descriptor is an integer of limited size, there is some limit to the depth of multiplexing that can be supported by the VuSystem. Being a 32-bit integer, the channel payload descriptor can store up to 2^{32} possible encodings. This is enough to support 2-port multiplexers nested up to a depth of 32, 3-port multiplexers nested up to a depth of 20, 4-port multiplexers nested up to a depth of 16, etc. Since these are quite deep nestings of multiplexers, a fixed `Channel` value of 32 bits should be adequate for all multiplexer configurations in any forseeable VuSystem application.

The `numInputPorts` VsMux Subcommand

```
<vsMux> numInputPorts [<numInputPorts>]
  ==> <numInputPorts>
```

The `numInputPorts` VsMux subcommand provides access to the numInputPorts parameter for a VsMux module, which specifies the number of input ports the module should have. It takes:

numInputPorts *(Integer)* A new number of input ports.

It returns:

numInputPorts *(Integer)* The current number of input ports.

A.12.8 The VsOrderedMerge Module

The VsOrderedMerge module merges two payload sequences into one. It works like the VsMerge module (page 153), except that it ensures that the payload timestamps in the output sequence are always increasing. It has any number of input ports, each named `input`N (for N from 0 to $numInputPorts - 1$); and one output port, named `output`. It is based on the VsEntity module.

The `numInputPorts` VsOrderedMerge Subcommand

```
<vsOrderedMerge> numInputPorts [<numInputPorts>]
  ==> <numInputPorts>
```

The `numInputPorts` VsOrderedMerge subcommand provides access to the numInputPorts parameter for a VsOrderedMerge module, which specifies the number of input ports the module should have. It takes:

numInputPorts *(Integer)* A new number of input ports.

It returns:

numInputPorts *(Integer)* The current number of input ports.

A.12.9 The VsOrderedMux Module

The VsOrderedMux module multiplexes many payload sequences into one. It is based on the VsEntity module. It works like the VsMux module (page 154), except that it ensures that the payload timestamps in the output sequence are always increasing. It has any number of input ports, each named `input`N (for N from 0 to $numInputPorts - 1$); and one output port, named `output`. Payload sequences that have been multiplexed by VsOrderedMux can be demultiplexed with VsDeMux (page 151). Multiplexing is performed by saving the input port number in the `channel` payload descriptor member of each payload.

A `VsMux` or `VsOrderedMux` module with n input ports records which input port from which payloads came, by updating the `Channel` payload descriptor so that $channel$ mod n returns which input port from which the payload came, and $channel/n$ returns the original `Channel` descriptor. A `VsDeMux` module with n output ports uses $channel$ mod n to select which output port to direct a payload, and replaces the `Channel` payload descriptor component with $channel/n$.

Since `Channel` payload descriptor is an integer of limited size, there is some limit to the depth of multiplexing that can be supported by the VuSystem. Being a 32-bit integer, the channel payload descriptor can store up to 2^{32} possible encodings. This is enough to support 2-port multiplexers nested up to a depth of 32, 3-port multiplexers nested up to a depth of 20, 4-port multiplexers nested up to a depth of 16, etc. Since these are quite deep nestings of multiplexers, a fixed `Channel` value of 32 bits should be adequate for all multiplexer configurations in any forseeable VuSystem application.

The `numInputPorts` VsOrderedMux Subcommand

```
<vsOrderedMux> numInputPorts [<numInputPorts>]
  ==> <numInputPorts>
```

The `numInputPorts` VsOrderedMux subcommand provides access to the numInputPorts parameter for a VsOrderedMux module, which specifies the number of input ports the module should have. It takes:

numInputPorts *(Integer)* A new number of input ports.

It returns:

numInputPorts *(Integer)* The current number of input ports.

A.12.10 The VsWipe Module

The VsWipe module performs a time-based wipe effect on video frames. All other payloads are passed transparently. It is based on the VsEffect module.

The VsWipe module indicates completion of the effect by calling its callback. The callback command string is evaluated with the following keyword parameter appended:

-done *(Boolean)* The effect has completed.

The `orientation` VsWipe Subcommand

```
<vsWipe> orientation [<orientation>]
  ==> <orientation>
```

The `orientation` VsWipe subcommand provides access to the orientation parameter for a VsWipe module. It takes:

orientation *(horizontal or vertical)* A new orientation.

It returns:

orientation *(horizontal or vertical)* The current orientation.

The `direction` VsWipe Subcommand

```
<vsWipe> direction [<direction>]
  ==> <direction>
```

The `direction` VsWipe subcommand provides access to the direction parameter for a VsWipe module. It takes:

direction *(forward or backward)* A new direction.

It returns:

direction *(forward or backward)* The current direction.

A.12.11 The VsTcpListener Module

The VsTcpListener module listens for new connection requests on a TCP port. When a connection request is received, it creates a VsTcpServer module (page 150) and calls its callback. It has no input or output ports. It is based on the VsEntity module.

The VsTcpListener module indicates through its callback that it has received a connection request and that it has created a VsTcpServer module (page 150). The callback command string is evaluated with the following keyword parameter appended:

-obj *(Command Name)* The object command for the VsTcpServer module.

The `backlog` VsTcpListener Subcommand

```
<vsTcpListener> backlog [<backlog>]
  ==> <backlog>
```

The `backlog` VsTcpListener subcommand provides access to the backlog parameter for a VsTcpListener module. It takes:

backlog *(Integer)* A new backlog.

It returns:

backlog *(Integer)* The current backlog.

The `port` VsTcpListener Subcommand

```
<vsTcpListener> port [<port>]
  ==> <port>
```

The `port` VsTcpListener subcommand provides access to the port parameter for a VsTcpListener module. It takes:

port *(Integer)* A new port.

It returns:

port *(Integer)* The current port.

The `Timeout` VsTcpListener Subcommand

```
<vsTcpListener> timeout [<timeout>]
  ==> <timeout>
```

The `Timeout` VsTcpListener subcommand provides access to the timeout parameter for a VsTcpListener module. It takes:

timeout *(Integer)* A new timeout.

It returns:

timeout *(Integer)* The current timeout.

Appendix B

Tcl Support Provided By The VuSystem

B.1 Vs Subcommands

The `appInitialize` Vs Subcommand

```
vs appInitialize <appContext> <name>
```

The `appInitialize` Vs subcommand initializes the VuSystem. It creates a top-level VsEntity object command and installs all the VsTclClass commands so that they may be used. It takes:

appContext *(Command Name)* A name of an appContext object command, created by the `xt appInitialize` (page 201) command or the `xt createApplicationContext` (page 201) command.

name *(String)* A name to use for the top-level object command to be created.

B.2 VsTclObj Subcommands

The `alias` VsTclObj Subcommand

```
<vsTclObj> alias <name>
```

The `alias` VsTclObj subcommand creates addtional object commands or aliases for an object. It takes:

name *(String)* A name to use for the object command to be created.

The `class` VsTclObj Subcommand

```
<vsTclObj> class [<name>]
  ==> <class>
```

The `class` VsTclObj subcommand provides access to the class object command for the object. It takes:

name *(String)* A name to use for the VsTclClass object command to be created, if it does not already exist.

It returns:

class *(Command Name)* The name of the vsTclClass object comand for this object's class.

The `info commands` VsTclObj Subcommand

```
<vsTclObj> info commands [<pattern>]
  ==> <commands>
```

The `info commands` VsTclObj subcommand returns a list of subcommands for the object. It takes:

pattern *(String)* A regular expression.

It returns:

commands *(List)* The list of command names that match the pattern. If no pattern is supplied, all command names are returned.

The `configCallback` VsTclObj Subcommand

```
<vsTclObj> configCallback [<command>]
  ==> <command>
```

The `configCallback` VsTclObj subcommand provides access to a command string that is executed whenever a change to the configuration of the object is made. Configuration changes include the creation or deletion of a child, connection or disconnection of a port, or any configuration change to a child. It takes:

command *(Command String)* A command string to be evaluated at configuration time.

It returns:

command *(Command String)* The configCallback command string.

The `destroy` VsTclObj Subcommand

```
<vsTclObj> destroy
```

The `destroy` VsTclObj subcommand destroys an object and deletes all object commands for the object. It also evaluates the `destroyCallback` (page 158) just before the object is destroyed.

The `destroyCallback` VsTclObj Subcommand

```
<vsTclObj> destroyCallback [<command>]
  ==> <command>
```

The `destroyCallback` VsTclObj subcommand provides access to the command string that is evaluated just before the object is destroyed. It takes:

command *(Command String)* A command string to be evaluated before destruction.

It returns:

command *(Command String)* The destroyCallback command string.

The `name` VsTclObj Subcommand

```
<vsTclObj> name
  ==> <name>
```

The `name` VsTclObj subcommand provides the primary object command name for the object. It returns:

name *(Command Name)* The primary object command name for this object.

The `names` VsTclObj Subcommand

```
<vsTclObj> names
  ==> <names>
```

The `names` VsTclObj subcommand provides access to the object command names for the object. It returns:

names *(List)* All object command names for this object.

The `options` VsTclObj Subcommand

```
<vsTclObj> info options [<pattern>]
  ==> <options>
```

The `info options` VsTclObj subcommand provides the option subcommand names for the object. It takes:

pattern *(String)* A regular expression.

It returns:

options *(List)* The list of option command names that match the pattern. If no pattern is supplied, all option command names are returned.

The `proc` VsTclObj Subcommand

```
<vsTclObj> proc <name> <args> <body>
```

The `proc` VsTclObj subcommand defines a new subcommand for the object. It is similar to the `proc` top-level command, except that the procedure is defined as a subcommand to the object, instead of as a top-level command. It takes:

name *(String)* A name for the proc.

args *(List)* A list of formal parameters to the proc.

args *(List)* A body for the proc. During evaluation of the procedure body, an additional local variable named `self` exists, whose value is the primary object command name of the object. In addition, any instance variables defined with the `set` (page 160) subcommands exist as local variables.

The `info procs` VsTclObj Subcommand

```
<vsTclObj> info procs [<pattern>]
  ==> <procs>
```

The `info procs` VsTclObj subcommand provides the procedure subcommand names for the object. It takes:

pattern *(String)* A regular expression.

It returns:

procs *(List)* The list of proc names that match the pattern. If no pattern is supplied, all proc names are returned.

The `rename` VsTclObj Subcommand

```
<vsTclObj> rename <from> <to>
```

The `rename` VsTclObj subcommand renames subcommands for the object. It takes:

old *(Command Name)* An old subcommand name.

new *(Command Name)* A new subcommand name.

The `set` VsTclObj Subcommand

```
<vsTclObj> set <name> [<value>]
  ==> <value>
```

The `set` VsTclObj subcommand provides access to instance variables for the object. These instance variables are also accessable to procedures defined with the `proc` (page 159) subcommand as local variables. It takes:

name *(String)* A name for an instance variable.

value *(String)* A value for the instance variable.

It returns:

value *(String)* The value of the instance variable.

The `info vars` VsTclObj Subcommand

```
<vsTclObj> info vars [<pattern>]
  ==> <vars>
```

The `info vars` VsTclObj subcommand provides the names of instance variables for the object. It takes:

pattern *(String)* A regular expression.

It returns:

vars *(List)* The list of variable names that match the pattern. If no pattern is supplied, all variable names are returned.

B.3 VsTclClass Subcommands

The `addInstance` VsTclClass Subcommand

```
<vsTclClass> addInstance <name>
```

The `addInstance` VsTclClass subcommand adds an object to its list of instances. It takes:

> **name** *(Command Name)* A name of an instance of this class.

The `classCommands` VsTclClass Subcommand

```
<vsTclClass> classCommands [<pattern>]
  ==> <classCommands>
```

The `classCommands` VsTclClass subcommand provides the names of subcommands defined for all instances of the class. It takes:

> **pattern** *(String)* A regular expression.

It returns:

> **classCommands** *(List)* The list of class command names that match the pattern. If no pattern is supplied, all class command names are returned.

The `classOptions` VsTclClass Subcommand

```
<vsTclClass> classOptions [<pattern>]
  ==> <classOptions>
```

The `classOptions` VsTclClass subcommand provides the names of option subcommands defined for all instances of the class. It takes:

> **pattern** *(String)* A regular expression.

It returns:

> **classOptions** *(List)* The list of class option command names that match the pattern. If no pattern is supplied, all class option command names are returned.

The `classProc` VsTclClass Subcommand

```
<vsTclClass> classProc <name> <args> <body>
```

The `classProc` VsTclClass subcommand defines a new subcommand for all instances of the class. It is similar to the `proc` top-level command, except that the procedure is defined as a subcommand to the instances, instead of as a top-level command. It is very similar to the `proc` (page 159) VsTclObj command, except that the procedure is defined for all instances of the class. It takes:

> **name** *(String)* A name for the proc.
>
> **args** *(List)* A list of formal parameters to the proc.
>
> **args** *(List)* A body for the proc. During evaluation of the procedure body, an additional local variable named `self` exists, whose value is the primary object command name of the instance. In addition, any instance variables defined with the `set` (page 160) subcommands exist as local variables.

The `classProcs` VsTclClass Subcommand

```
<vsTclClass> classProcs [<pattern>]
  ==> <classProcs>
```

The `classProcs` VsTclClass subcommand provides the names of procedure commands for all instances of the class. It takes:

pattern *(String)* A regular expression.

It returns:

classProcs *(List)* The list of class proc names that match the pattern. If no pattern is supplied, all class proc names are returned.

The `create` VsTclClass Subcommand

```
<vsTclClass> create <name> [<keyword> <value>]...
```

The `create` VsTclClass subcommand creates an instance of the class. It takes:

name *(String)* A name for the instance to be created.

keyword *(String)* A name of an option command the instance provides.

value *(String)* A value for the option command.

The `instances` VsTclClass Subcommand

```
<vsTclClass> instances [<pattern>]
  ==> <instances>
```

The `instances` VsTclClass subcommand returns the object command names of instances of the class. It takes:

pattern *(String)* A regular expression.

It returns:

instances *(List)* The list of instance names that match the pattern. If no pattern is supplied, all instance names are returned.

The `removeInstance` VsTclClass Subcommand

```
<vsTclClass> removeInstance <name>
```

The `removeInstance` VsTclClass subcommand removes an object from its list of instances. It takes:

name *(Command Name)* A name of an instance of this class.

The `superClass` VsTclClass Subcommand

```
<vsTclClass> superClass [<parent>]
  ==> <parent>
```

The **`superClass`** VsTclClass subcommand provides access to the superclass of the class. It takes:

> **parent** *(Command Name)* A VsTclClass name to set the superclass to this class.

It returns:

> **parent** *(Command Name)* The name of the superclass to this class.

B.4 VsEntity Subcommands

The `callback` VsEntity Subcommand

```
<vsEntity> callback [<command>]
  ==> <command>
```

The **`callback`** VsEntity subcommand provides access to the callback command string for the object. It takes:

> **command** *(Command String)* A command string to be evaluated at event times.

It returns:

> **command** *(Command String)* The callback command string.

The `children` VsEntity Subcommand

```
<vsEntity> children
  ==> <children>
```

The **`children`** VsEntity subcommand provides the object command names of all the children of this object. It returns:

> **children** *(List)* The children of this module.

The `inputs` VsEntity Subcommand

```
<vsEntity> inputs
  ==> <inputs>
```

The **`inputs`** VsEntity subcommand provides the object command names for all input ports of the object. It returns:

> **inputs** *(List)* The input ports of this module.

The `outputs` VsEntity Subcommand

```
<vsEntity> outputs
  ==> <outputs>
```

The `outputs` VsEntity subcommand provides the object command names for all the output ports of the object. It returns:

> **outputs** *(List)* The output ports of this module.

The `start` VsEntity Subcommand

```
<vsEntity> start [<mode>]
```

The `start` VsEntity subcommand starts the object. It causes the `Start` (page 175) member functions for this object and all its children to be called, which start in-band processing. It takes:

> **mode** *(Boolean)* 1 to cause source modules to send `VsStart` payloads, 0 otherwise.

The `stop` VsEntity Subcommand

```
<vsEntity> stop [<mode>]
```

The `stop` VsEntity subcommand stops the object. It causes the `Stop` (page 176) member functions for this object and all its children to be called, which stop in-band processing. It takes:

> **mode** *(Boolean)* 1 to cause modules to wait for `VsFinish` payloads before shutting down, 0 to shut down immediately.

The `xPosition` VsEntity Subcommand

```
<vsEntity> xPosition [<xPosition>]
  ==> <xPosition>
```

The `xPosition` VsEntity subcommand provides access to the objects x position. It is used by a grapical programming system under development. It takes:

> **xPosition** *(Integer)* A new x position.

It returns:

> **xPosition** *(Integer)* The x position.

The `yPosition` VsEntity Subcommand

```
<vsEntity> yPosition [<yPosition>]
  ==> <yPosition>
```

The `yPosition` VsEntity subcommand provides access to the objects y position. It is used by a graphical programming system under development. It takes:

> **xPosition** *(Integer)* A new x position.

It returns:

> **xPosition** *(Integer)* The x position.

B.5 VsInputPort Subcommands

The `bind` VsInputPort Subcommand

```
<vsInputPort> bind [<outputPort>]
  ==> <outputPort>
```

The `bind` VsInputPort subcommand associates an output port with the input port. It takes:

> **outputPort** *(Command Name)* An output port to bind.

It returns:

> **outputPort** *(Command Name)* The output port currently bound.

The `unbind` VsInputPort Subcommand

```
<vsInputPort> unbind <outputPort>
```

The `unbind` VsInputPort subcommand disassociates an output port with the input port. It takes:

> **outputPort** *(Command Name)* An output port to unbind.

B.6 VsOutputPort Subcommands

The `connect` VsOutputPort Subcommand

```
<vsOutputPort> connect [<inputPort>]
  ==> <inputPort>
```

The `connect` VsOutputPort subcommand associates an input port with the output port. It takes:

> **inputPort** *(Command Name)* An input port to connect.

It returns:

> **inputPort** *(Command Name)* The input port currently connected.

The `disconnect` VsOutputPort Subcommand

```
<vsOutputPort> disconnect <inputPort>
```

The `disconnect` VsOutputPort subcommand disassociates an input port with the output port. It takes:

> **inputPort** *(Command Name)* An input port to disconnect.

B.7 Utility Commands

The following are some useful commands that are not part of standard Tcl distribution, but are useful to VuSystem application scripts.

The `date` Command

```
date [<format>]
  ==> <date>
```

Use the `date` command to get the current date and time in a formatted string. It uses the `strftime` POSIX procedure. It takes:

> **format** *(String)* The format control string. Characters are copied from this string to the result, with substitutions occuring whenever the % character is encountered. The character following the % character is used to specify the substitution. See the `strftime` POSIX procedure description for the meanings of the substitution characters. The default value of the `format` argument is "%c".

It returns:

> **date** *(String)* The current date and time formatted as specified.

The `sleep` Command

```
sleep <seconds>
```

Use the `sleep` command to sleep for a certain amount of time. It takes:

> **seconds** *(Float)* A floating point value indicating the number of seconds to sleep.

The `true` Command

```
true
  ==> 1
```

The `true` command always returns 1. Use it instead of 1 where a boolean value is necessary, to enhance readability of a script.

The `false` Command

```
false
  ==> 0
```

The `false` command always returns 0. Use it instead of 0 where a boolean value is necessary, to enhance readability of a script.

B.8 Commands To Support The Manipulation Of Keyword Argument Lists

Many VuSystem scripts use keyword argument lists to pass parameters around. The `keyarg` (page 167), `keyargs` (page 167), and `apply` (page 168) commands are useful to these scripts.

The `keyarg` Command

```
keyarg <keyword> <args> [<default>] [<required>]
  ==> <value>
```

Use the **keyarg** command extract the value of a keyword-specified argument from a list of alternating keywords and values. It compares every other element of a list with a specified keyword, and if the keyword matches, it returns the next element in the list. It takes:

keyword *(String)* The keyword to match in the argument list.

args *(List)* The argument list to search.

default *(String)* The value to return if the keyword could not be found in the list.

required *(String)* If this argument is the word `required`, an error is signaled if the keyword could not be found in the list.

It returns:

value After comparing every other element of `args` with `keyword`, if a match was found, then the next element in the list. Otherwise `default`.

The `keyargs` Command

```
keyargs <keywords> <args> [<exclude>]
  ==> <args>
```

Use the **keyargs** command to create new argument lists from existing argument lists. It extracts keyword value pairs from a list of alternating keywords and values. It compares every other element of a list with a specified keyword, and if the keyword matches, it returns the next element in the list. It takes:

keywords *(List)* A list of keywords specifying which values to extract from the argument list. Members of the `keywords` list can either be keywords, or pairs of keywords:

- If a member of the keyword list is a single keyword, then it is used as a search key for the supplied argument list, and also as a specification keyword in the result argument list.
- If a member of the keyword list is a pair of keywords, then the first keyword is used as the search key for the supplied argument list, and the second keyword is used as the specification keyword in the result argument list.

args *(List)* The argument list to search.

exclude *(String)* If this argument is the word `exclude`, then the result list is a list of keyword value pairs whose keywords do *not* match any keyword in the keywords list. This is useful for building argument lists with certain arguments removed from them.

It returns:

args *(List)* The new keyword argument list.

The apply Command

```
apply [<command> [<arg>...]] <args>
  ==> <result>
```

Use the apply command to invoke a command for which you have some of the argument list in list form. It is especially handy if a keyword argument list is being supplied to a command. It takes:

command *(Command Name)* The name of the command to invoke. If it is not supplied, then the first value in the args list is taken to be the command. Members of the keywords list can either be keywords, or pairs of keywords:

arg *(String)* Any number of arguments to the command.

args *(List)* The rest of the arguments to the command. Effectively, this list is *spread* out over the command, instead of supplied as a single argument.

It returns:

result *(String)* The result of command invocation.

B.9 Commands To Support The Interactive Entry Of Tcl Commands

Some applications might have a special window where Tcl Commands can be typed in by the user. The assemble and assembleDestroy commands are useful to support this.

The assemble Command

```
assemble <bufName> <commandPiece>
  ==> <cmd>
```

Use the assemble command to incrementally assemble tcl commands before having them evaluated. When a complete command has been assembled, it is returned. The assemble is useful if you are implementing a graphical user interface that supports the typing of commands. It takes:

bufName *(Handle)* The name of the command buffer in which to assemble the command. This allows multiple independent commands to be assembled simultaneously.

commandPiece *(String)* The string to be appended to the command buffer.

It returns:

cmd *(Command String)* The command string if a complete one has been assembled, otherwise the empty string.

The assembleDestroy Command

```
assembleDestroy <bufName>
```

Use the assembleDestroy command to destroy an assemble buffer. It takes:

bufName *(Handle)* The name of the command buffer to destroy.

B.10 Commands To Support Debugging And Low-Level Operations

Some VuSystem applications are used to manipulate low-level interfaces. For example, the **vuptest** and **vudtest** VuSystem applications are used to test network device drivers. The following are some commands useful to such applications.

The lsbFirst Command

```
lsbFirst
  => <lsbFirst>
```

Use the **lsbFirst** command whenever your script needs to know the endianness of the machine on which you are running. It returns:

> **lsbFirst** *(Boolean)* 1 if the machine is a little-endian machine, and 0 if it is a big-endian machine.

The debug Command

```
debug [<mode>]
  ==> <mode>
```

Use the **debug** command to set the debug mode. It always returns the value of the current debug mode, and sets it if an argument is supplied. It takes:

> **mode** *(Integer)* An integer value to set the debug mode. It is an integer value that is used to enable the execution of debugging code throughout the VuSystem shell. In a VuSystem shell that has been compiled for debugging, certain bits in the debug mode turn on certain print statements.

It returns:

> **mode** *(Integer)* The current value of the debug mode.

Appendix C

Support For Modules In The VuSystem

C.1 Sending Data To A Downstream Module

The Send VsOutputPort Member Function

```
Boolean
Send(VsPayload* payload);
```

Use the `Send` VsOutputPort member function to send data through the port to a downstream module. It returns `True` if the downstream module accepted the payload. If it returns `False`, you should try again later, in an `Idle` (page 171) member function. It takes:

payload The payload to be sent.

The Idle Member Function

```
virtual void
Idle(VsOutputPort* outputPort);
```

Implement an `Idle` member function if your module has any output ports and you are not subclassing `VsFilter` (page 180). It is called whenever any downstream module may be ready for more data. It should use the `Send` (page 171) VsOutputPort member function to send any payloads that can be sent. An `Idle` member function returns no values and takes:

outputPort The outputPort which may be ready for more data.

C.2 Receiving Data From An Upstream Module

The Idle VsInputPort Member Function

```
void
Idle();
```

Use the `Idle` VsInputPort member function to indicate when your module is ready for more data. Additional data will arrive though the `Receive` (page 172) member function. The `Idle` VsInputPort member function returns no values and takes no arguments.

The `Receive` Member Function

```
virtual Boolean
Receive(VsInputPort* inputPort, VsPayload* payload);
```

Implement a `Receive` member function if your module has any input ports and you are not subclassing `VsFilter` (page 180). It is called whenever any upstream module has data to be sent. It should return `True` if the payload is accepted, and `False` if the module is not ready for the payload and you want the upstream module to try again later. Some time after returning `False`, your module should use the `Idle` (page 171) VsInputPort member function to indicate that the module is ready for more data. A `Receive` member function takes:

inputPort The inputPort from which the data is being sent.

payload The data.

C.3 Scheduling Computation Operations

The `Work` Member Function

```
virtual Boolean
Work();
```

Implement a `Work` member function if your module is to do any substantial computation. Once started with the `StartWork` (page 172) member function, it is called regularly until it either returns `True` or it is stopped with the `StopWork` (page 172) member function. A `Work` member function takes no arguments.

The `StartWork` Member Function

```
VsWorkId
StartWork();
```

Use the `StartWork` member function to indicate that you want the `Work` (page 172) member function to be called. The `StartWork` member function takes no values and returns a work identifier that should be saved to be used in any subsequent calls to the `StopWork` (page 172) member function.

The `StopWork` Member Function

```
void
StopWork(VsWorkId workId);
```

Use the `StopWork` member function to stop calls to the `Work` (page 172) member function. The `StopWork` member function returns no values and takes:

workId The work identifier returned from the `StartWork` (page 172) member function.

C.4 Scheduling Time-Dependent Operations

The Timeout Member Function

```
virtual void
timeout(VsIntervalId intervalId);
```

Implement a `Timeout` member function if your module has any operations that should be performed at a particular time. It is called at a time indicated through the `StartTimeout` (page 173) member function. It returns no values and takes:

intervalId The interval identifier returned from the `StartTimeout` (page 173) member function.

The StartTimeout Member Function

```
VsIntervalId
StartTimeout(const VsTimeval& time);
```

Use the `StartTimeout` member function to indicate that you want the `Timeout` (page 173) member function to be called at a particular time. The `StartTimeout` member function returns the interval identifier that should be saved to be used in any subsequent calls to the `StopTimeout` (page 173) member function. The `StartTimeout` member function takes:

time An absolute time when the `Timeout` member function should be called.

The StopTimeout Member Function

```
void
StopTimeout(VsIntervalId intervalId);
```

Use the `StopTimeout` member function to stop a call to the `timeout` (page 173) member function. The `StopTimeout` returns no values and takes:

intervalId The interval identifier returned from the `StartTimeout` member function.

VsTimeval Values

```
class VsTimeval : public timeval {
  ...
public:
  VsTimeval();
  VsTimeval(long usec);
  VsTimeval(long sec, long usec);
  VsTimeval operator+(const VsTimeval&) const;
  VsTimeval operator-(const VsTimeval&) const;
  VsTimeval operator*(double) const;
  VsTimeval& operator+=(const VsTimeval&);
  VsTimeval& operator-=(const VsTimeval&);
  VsTimeval& operator*=(double);
  int operator==(const VsTimeval&) const;
  int operator!=(const VsTimeval&) const;
  int operator>=(const VsTimeval&) const;
  int operator<=(const VsTimeval&) const;
  int operator>(const VsTimeval&) const;
  int operator<(const VsTimeval&) const;
  long Milliseconds() const;
  long Microseconds() const;
  static int Get(Tcl_Interp*, char*, VsTimeval*);
  int Return(Tcl_Interp*) const;
  static VsTimeval Now();
};
```

VsTimeval values are used to represent absolute time values with microsecond precision. The `StartTimeout` (page 173) member function takes a VsTimeval value to designate when the `Timeout` (page 173) member function should be called.

VsTimeval member functions provide facilities for adding to, subtracting from, scaling, and comparing VsTimeval values. Of particular interest is the `Now` VsTimeval static member function, which returns a VsTimeval value corresponding to the current time.

C.5 Scheduling File Input Operations.

The `Input` Member Function

```
virtual void
Input(int fileDescriptor, VsInputId inputId);
```

Implement an `Input` member function if your module performs input operations on files. It is called whenever a file indicated with the `StartInput` (page 174) member function is ready for input. An `Input` member function returns no values and takes:

fileDescriptor The Unix file descriptor for the file.

inputId The input identifier returned from the `StartInput` (page 174) member function.

The `StartInput` Member Function

```
VsInputId
StartInput(int fileDescriptor);
```

Use the `StartInput` member function to indicate that you want the `Input` (page 174) member function to be called whenever a particular file is ready for input. The `StartInput` member function returns an input identifier that should be saved to be used in any subsequent calls to the `StopInput` (page 174) member function. The `StartInput` member function takes:

fileDescriptor The Unix file descriptor for the file.

The `StopInput` Member Function

```
void
StopInput(VsInputId inputId);
```

Use the `StopInput` member function to stop calls to the `Input` (page 174) member function. The `StopInput` member function returns no values and takes:

inputId The input identifier returned from the `StartInput` (page 174) member function.

C.6 Scheduling File Output Operations.

The `Output` Member Function

```
virtual void
Output(int fileDescriptor, VsOutputId outputId);
```

Implement an `Output` member function if your module performs output operations on files. It is called whenever a file indicated with the `StartOutput` (page 175) member function is ready for output. An `Output` member function returns no values and takes:

fileDescriptor The Unix file descriptor for the file.

outputId The output identifier returned from the `StartOutput` (page 175) member function.

The `StartOutput` Member Function

```
VsOutputId
StartOutput(int fileDescriptor);
```

Use the `StartOutput` member function to indicate that you want the `Output` (page 175) member function to be called whenever a particular file is ready for output. The `StartOutput` member function returns an output identifier that should be saved to be used in any subsequent calls to the `StopOutput` (page 175) member function. The `StartOutput` member function takes:

fileDescriptor The Unix file descriptor for the file.

The `StopOutput` Member Function

```
void
StopOutput(VsOutputId outputId);
```

Use the `StopOutput` member function to stop calls to the `Output` (page 175) member function. The `StopOutput` member function returns no values and takes:

outputId The output identifier returned from the `StartOutput` (page 175) member function.

C.7 Starting and Stopping

The `Start` Member Function

```
virtual void
Start(Boolean mode);
```

Implement a `Start` member function if your module needs to perform any operations at the beginning of in-band processing. This would include any initial calls to the `StartInput` (page 174) and `StartTimeout` (page 173) member functions. A `Start` member function returns no values and takes:

mode The start mode. If `False`, then source modules should send a `VsStart` (page 195) payload to mark a synchronous starting point in the data stream after starting.

The `Stop` Member Function

```
virtual void
Stop(Boolean mode);
```

Implement a `Stop` member function if your module needs to perform any operations at the end of in-band processing. This includes any final calls to the `StopWork` (page 172), `StopTimeout` (page 173), `StopInput` (page 174), and `StopOutput` (page 175) member functions. A `Stop` member function returns no values and takes:

mode The stop mode. If `False`, then source modules should send a `VsFinish` (page 192) payload to mark a synchronous stopping point in the data stream before stopping, and all other modules should prepare to stop when they receive a `Finish` payload. If `True` then all modules should stop immediately.

C.8 Adding Tcl Subcommands

The `CreateCommand` Member Function

```
void
CreateCommand(char* commandName,
              Tcl_CmdProc* commandProc,
              ClientData clientData,
              Tcl_CmdDeleteProc* deleteProc,
              char* documentation = "",
              CommandType type = VSCOMMAND);
```

Use the `CreateCommand` member function in the constructor function for your module class to register subcommand procedures. The `CreateCommand` member function returns no values and takes:

commandName The name of the subcommand.

commandProc The friend procedure that implements the subcommand.

clientData A pointer to the module ("`this`").

deleteProc A procedure to be called when the subcommand is deleted.

documentation A short documentation string describing the subcommand.

type One of `VSCOMMAND` or `VSOPTIONCOMMAND`.

The `CreateOptionCommand` Member Function

```
void
CreateOptionCommand(char* commandName,
                    Tcl_CmdProc* commandProc,
                    ClientData clientData = 0,
                    Tcl_CmdDeleteProc* deleteProc = 0,
                    char* documentation = "");
```

Use the `CreateOptionCommand` member function in the constructor function for your module class to register option subcommand procedures. The `CreateOptionCommand` member function returns no values and takes:

commandName The name of the subcommand.

commandProc The friend procedure that implements the subcommand.

clientData A pointer to the module ("`this`").

deleteProc A procedure to be called when the subcommand is deleted.

documentation A short documentation string describing the subcommand.

C.9 Calling Tcl Callbacks

The `EvalCallback` Member Function

```
void
EvalCallback(char* args);
```

Use the **EvalCallback** member function in modules to call Tcl callbacks, supplying a parameter string describing what event has occurred. The parameter string can be used to provide event-specific parameters, or to indicate which event has occurred if more than one type of event may be signalled. The **EvalCallback** member function returns no values and takes:

args A character string to be appended to the command provided by the application programmer before the whole string is evaluated.

C.10 Initialization

The `classSymbol` Static Variable

```
VsSymbol* YourClass::classSymbol;
```

Implement a **classSymbol** static variable to provide a place for storing a class name. Examples:

```
VsSymbol* VsVidboardSource::classSymbol;

VsSymbol* VsPuzzle::classSymbol;
```

The `ObjPtr` Member Function

```
virtual void*
ObjPtr(const VsSymbol* cl);
```

Implement an **ObjPtr** member function to provide a facility for checking whether an instance of your class is an instance of a specified class. An **ObjPtr** member function returns a pointer to an instance and takes:

cl The class symbol representing the class for which a pointer is requested.

Examples:

```
void*
VsVidboardSource::ObjPtr(const VsSymbol* cl) {
  return (cl == classSymbol)? this : VsEntity::ObjPtr(cl);
}

void*
VsPuzzle::ObjPtr(const VsSymbol* cl) {
  return (cl == classSymbol)? this : VsFilter::ObjPtr(cl);
}
```

The `DerivePtr` Static Member Function

```
static YourClass*
DerivePtr(VsObj* o);
```

Implement a `DerivePtr` static member function to provide a facility for converting from a pointer to a `VsObj` instance to a pointer to an instance of your class. It is typically defined as a inline member function, since it is so short. A `DerivePtr` member function returns a pointer to an instance of the class and takes:

o A pointer to a `VsObj` instance.

Examples:

```
inline VsVidboardSource*
VsVidboardSource::DerivePtr(VsObj* o) {
  return (VsVidboardSource*)o->ObjPtr(classSymbol);
}

inline VsPuzzle*
VsPuzzle::DerivePtr(VsObj* o) {
  return (VsPuzzle*)o->ObjPtr(classSymbol);
}
```

The `Get` Static Member Function

```
static int
Get(Tcl_Interp* in, char* nm, YourClass** pp);
```

Implement a `Get` static member function to provide a facility for converting from a Tcl command name, to a pointer to an instance of your class. It is typically defined as an inline member function, since it is so short. A `Get` member function returns a Tcl status and takes:

in The pointer to the Tcl interpreter.

nm The pointer to the Tcl command name.

pp The pointer to the location to store the instance pointer.

Examples:

```
inline int
VsVidboardSource::Get(Tcl_Interp* in,char* nm,VsVidboardSource** pp) {
  return VsTclObj::Get(in,nm,classSymbol,(void**)pp);
}

inline int
VsPuzzle::Get(Tcl_Interp* in, char* nm, VsPuzzle** pp) {
  return VsTclObj::Get(in, nm, classSymbol, (void**)pp);
}
```

The `Creator` Static Member Function

```
static VsEntity*
Creator(Tcl_Interp* in,VsEntity* pr,const char* nm);
```

Implement a `Creator` static member function to provide a facility for creating instances of your class. A `Creator` member function returns a pointer to an instance of your class and takes:

in The pointer to the Tcl interpreter.

pr The pointer to the parent module.

nm A pointer to the child name for this instance.

Examples:

```
VsEntity*
VsVidboardSource::Creator(Tcl_Interp* in,VsEntity* pr,const char* nm) {
  return new VsVidboardSource(in,pr,nm);
}

VsEntity*
VsPuzzle::Creator(Tcl_Interp* in,VsEntity* pr,const char* nm) {
  return new VsPuzzle(in,pr,nm);
}
```

The `InitClass` Static Member Function

```
static VsSymbol*
InitClass(Tcl_Interp* interp,
          VsEntityCreatorProc* creator,
          char* name,
          char* superClass);
```

Use the `InitClass` static member function to install your module. It returns a pointer to a symbol which should be saved in a `classSymbol` (page 177) class variable for use by the `ObjPtr` (page 177) and `DerivePtr` (page 178) member functions. The `InitClass` static member function takes:

interp The Tcl interpreter.

creator A procedure that creates a new instance of the module.

name The name of the module class.

superClass The name of the superclass of the module class.

The `InitInterp` static member function

```
static void
InitInterp(Tcl_Interp* interp);
```

Implement an `InitInterp` static member function that installs your module by calling the `InitClass` static member function. An `InitInterp` static member function returns no values and takes:

interp The Tcl interpreter.

Examples:

```
void
VsVidboardSource::InitInterp(Tcl_Interp* in) {
  classSymbol = InitClass(in,Creator,"VsVidboardSource","VsEntity");
}

void
VsPuzzle::InitInterp(Tcl_Interp* in) {
  classSymbol = InitClass(in,Creator,"VsPuzzle","VsFilter");
}
```

C.11 Filter Modules

If the module you are designing has one input port and one output port, you are designing a *filter* module. If your filter is simply computation-based, you should subclass the `VsFilter` class, and implement a `WorkRequiredP` (page 180) member function and a `Work` (page 180) member function. There is no need to implement an `Idle` (page 171) or `Receive` (page 172) member function.

The `WorkRequiredP` Member Function

```
virtual Boolean
WorkRequiredP(VsPayload *p) {
```

Implement a `WorkRequiredP` member function to return `True` if the work function should be called for this payload, and `False` if the payload should just be passed on without processing. Example:

```
Boolean
VsPuzzle::WorkRequiredP(VsPayload *p) {
  return solved == False && VsVideoFrame::DerivePtr(p) != 0;
}
```

The `Work` Member Function

```
virtual Boolean
Work();
```

Implement a `Work` member function to perform your filter computation. The input to your computation is in the `payload` instance variable. Perform the computation and put the result in the `payload` instance variable. Finally, call `VsFilter::Work()` and return its result. Example:

```
VsPuzzle::Work() {
  VsVideoFrame* frame = VsVideoFrame::DerivePtr(payload);
  if (!solved) {
    VsXdrBlock newData(frame->Data().Fore());
    /* scramble the image */
    frame->Data() = newData;
  }
  return VsFilter::Work();
}
```

C.12 Signalling and Handling Errors

The `VsPanic` Procedure

```
void
VsPanic(const char* msg, ...);
```

Use the `VsPanic` procedure to report fatal errors to the user, supplying a string and up to 10 additional parameters suitable for `printf`. `VsPanic` aborts execution of the program and takes:

msg A character string suitable for `printf`.

... Additional parameters suitable for `printf`.

Example:

```
if (mustBeZero != 0) VsPanic("%s: I can not take it any longer.",
Name());
```

The VsError Procedure

```
void
VsError(const char* msg, ...);
```

Use the **VsError** procedure to queue error messages in **VsErrRec** structures for later reporting to the user, supplying a string and up to 10 additional parameters suitable for **sprintf**. **VsError** does not interrupt the flow of execution of the program. It only reports errors. You still need to recover from the error condition. **VsError** returns no values and takes:

msg A character string suitable for **sprintf**.

... Additional parameters suitable for **sprintf**.

Example:

```
if (x < 0.0) {
  VsError("%s: The peasants are revolting", Name());
  y = 0;
} else y = sqrt(x);
```

The VsPushErrRec Procedure

```
void
VsPushErrRec(VsErrRec *erPtr);
```

Use the **VsPushErrRec** procedure to start a section of code where errors reported with **VsError** are captured and reported to the user. **VsPushErrRec** clears a VsErrRec structure and pushes it on to a stack. **VsPushErrRec** returns no values and takes:

erPtr A pointer to a **VsErrRec** structure.

Example:

```
{
  VsErrRec rec; VsPushErrRec(&rec);
  src->Stop(False);
  src->Start(False);
  if (VsPopErrRec(&rec)) return VsErrRecToTclErr(in, &rec);
}
```

The VsPopErrRec Procedure

```
int
VsPopErrRec(VsErrRec *erPtr);
```

Use the **VsPopErrRec** procedure to end a section of code where errors reported with **VsError** are captured and reported to the user. **VsPopErrRec** pops a **VsErrRec** structure off a stack and checks whether any errors reported with **VsError** have been queued on it. **VsPopErrRec** returns 1 if any errors have been queued, and 0 if not. It takes:

erPtr A pointer to a **VsErrRec** structure.

Example:

```
{
  VsErrRec rec; VsPushErrRec(&rec);
  src->Stop(False);
  src->Start(False);
  if (VsPopErrRec(&rec)) return VsErrRecToTclErr(in, &rec);
}
```

The VsErrRecToTclErr Procedure

```
int
VsErrRecToTclErr(Tcl_Interp *interp,
                 VsErrRec *erPtr);
```

Use the VsErrRecToTclErr procedure to convert any errors reported with VsError and captured into a VsErrRec structure to a Tcl error message. VsErrRecToTclErr converts error messages queued into a VsErrRec structure by VsError to Tcl error messages. VsErrRecToTclErr returns TCL_ERROR if any error messages were converted, and TCL_OK if not. It takes:

interp A pointer to the Tcl interpreter in which the errors should be signalled.

erPtr A pointer to a VsErrRec structure.

Example:

```
{
  VsErrRec rec; VsPushErrRec(&rec);
  src->Stop(False);
  src->Start(False);
  if (VsPopErrRec(&rec)) return VsErrRecToTclErr(in, &rec);
}
```

The VsTclErrArgCnt Procedure

```
int
VsTclErrArgCnt(Tcl_Interp *interp,
               char *cmdname,
               char *arglist);
```

Use the VsTclErrArgCnt procedure to report an incorrect number of parameters to a Tcl command procedure. VsTclErrArgCnt sets the Tcl result string to

```
wrong # args: should be {CMDNAME ARGLIST}
```

where CMDNAME is specified by the cmdname parameter and ARGLIST by the arglist parameter. It returns TCL_ERROR and takes:

interp A pointer to the Tcl interpreter in which the error should be signalled.

cmdname The name of the current Tcl command. This string is used in the error message.

arglist A description of the formal parameters to the Tcl command. This string is used in the error message.

Example:

```
int
SimpleFileSourceSourcePathnameCmd(ClientData cd, Tcl_Interp* in, int argc,
                                  char* argv[])
{
  SimpleFileSource* src = (SimpleFileSource*)cd;
  if (argc > 2) return VsTclErrArgCnt(in, argv[0], "?pathname?");
  ...
  return VsReturnString(in, src->pathname, TCL_STATIC);
}
```

The VsTclErrBadVal Procedure

```
int
VsTclErrBadVal(Tcl_Interp *interp,
               char *expected,
               char *value);
```

Use the **VsTclErrBadVal** procedure to report a bad value for a parameter to a Tcl command procedure. **VsTclErrBadVal** sets the result Tcl result string to

```
expected EXPECTED but got VALUE
```

where EXPECTED is specified by the expected parameter and VALUE by the value parameter. It returns TCL_ERROR and takes:

interp A pointer to the Tcl interpreter in which the error should be signalled.

expected A description of what was expected. This string is used in the error message.

value The actual parameter value. This string is used in the error message.

Example:

```
int
VsPuzzlePositionCmd(ClientData cd,Tcl_Interp* in,int argc,char* argv[]){
  ...
  if (argc == 2) {
    int x, y;
    if (VsGetIntPair(in, argv[1], &x, &y) != TCL_OK) return TCL_ERROR;
    if (x >= p->dim || x < 0)
      return VsTclErrBadVal(in, "x position within range", argv[1]);
    if (y >= p->dim || y < 0)
      return VsTclErrBadVal(in, "y position within range", argv[2]);
    if (x-p->x != 1 && x-p->x != -1 && y-p->y != 1 && y-p->y != -1)
      return VsTclErrBadVal(in, "x or y adjacent", "none");
    if (x-p->x != 0 && y-p->y != 0)
      return VsTclErrBadVal(in, "x or y adjacent", "both");
  ...
  }
  ...
}
```

C.13 Tcl Command Input Parameter Parsing

The VsGetBoolean Procedure

```
int
VsGetBoolean(Tcl_Interp *interp,
             String val,
             Boolean *ptr);
```

Use the VsGetBoolean procedure to convert an input parameter to a Boolean. If the input parameter conversion was successful, VsGetBoolean stores the data where ptr points and returns TCL_OK. If an error occurred during conversion, VsGetBoolean stores nothing where ptr points and returns TCL_ERROR. The VsGetBoolean procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

ptr A pointer to where the result should go.

The `VsGetChar` Procedure

```
int
VsGetChar(Tcl_Interp *interp,
          String val,
          char *ptr);
```

Use the `VsGetChar` procedure to convert an input parameter to a `char`. If the input parameter conversion was successful, `VsGetChar` stores the data where `ptr` points and returns `TCL_OK`. If an error occurred during conversion, `VsGetChar` stores nothing where `ptr` points and returns `TCL_ERROR`. The `VsGetChar` procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

ptr A pointer to where the result should go.

The `VsGetFloat` Procedure

```
int
VsGetFloat(Tcl_Interp *interp,
           String val,
           float *ptr);
```

Use the `VsGetFloat` procedure to convert an input parameter to a `float`. If the input parameter conversion was successful, `VsGetFloat` stores the data where `ptr` points and returns `TCL_OK`. If an error occurred during conversion, `VsGetFloat` stores nothing where `ptr` points and returns `TCL_ERROR`. The `VsGetFloat` procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

ptr A pointer to where the result should go.

The `VsGetFloatPair` Procedure

```
int
VsGetFloatPair(Tcl_Interp *interp,
               String val,
               float *xptr,
               float *yptr);
```

Use the `VsGetFloatPair` procedure to convert an input parameter to a `float` pair. If the input parameter conversion was successful, `VsGetFloatPair` stores the data where `xptr` and `yptr` point and returns `TCL_OK`. If an error occurred during conversion, `VsGetFloatPair` stores nothing where `xptr` and `yptr` point and returns `TCL_ERROR`. The `VsGetFloatPair` procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

xptr A pointer to where the first result of the pair should go.

yptr A pointer to where the second result of the pair should go.

The VsGetDouble Procedure

```
int
VsGetDouble(Tcl_Interp *interp,
            String val,
            double *ptr);
```

Use the VsGetDouble procedure to convert an input parameter to a double. If the input parameter conversion was successful, VsGetDouble stores the data where ptr points and returns TCL_OK. If an error occurred during conversion, VsGetDouble stores nothing where ptr points and returns TCL_ERROR. The VsGetDouble procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

ptr A pointer to where the result should go.

The VsGetInt Procedure

```
int
VsGetInt(Tcl_Interp *interp,
         String val,
         int *ptr);
```

Use the VsGetInt procedure to convert an input parameter to an int. If the input parameter conversion was successful, VsGetInt stores the data where ptr points and returns TCL_OK. If an error occurred during conversion, VsGetInt stores nothing where ptr points and returns TCL_ERROR. The VsGetInt procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

ptr A pointer to where the result should go.

The VsGetIntPair Procedure

```
int
VsGetIntPair(Tcl_Interp *interp,
             String val,
             int *xptr,
             int *yptr);
```

Use the VsGetIntPair procedure to convert an input parameter to an int pair. If the input parameter conversion was successful, VsGetIntPair stores the data where xptr and yptr point and returns TCL_OK. If an error occurred during conversion, VsGetIntPair stores nothing where xptr and yptr point and returns TCL_ERROR. The VsGetIntPair procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

xptr A pointer to where the first result of the pair should go.

yptr A pointer to where the second result of the pair should go.

The `VsGetIntList` Procedure

```
int
VsGetIntList(Tcl_Interp *interp,
             String val,
             int *cptr,
             int **lptr);
```

Use the `VsGetIntList` procedure to convert an input parameter to an `int` list. If the input parameter conversion was successful, `VsGetIntList` stores the list length where `cptr` points and the list data where `lptr` points and returns `TCL_OK`. If an error occurred during conversion, `VsGetIntList` stores nothing where `cptr` and `lptr` point and returns `TCL_ERROR`. The `VsGetIntList` procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

cptr A pointer to where the result list length should go.

lptr A pointer to where the result list data should go.

The `VsGetLong` Procedure

```
int
VsGetLong(Tcl_Interp *interp,
          String val,
          long *ptr);
```

Use the `VsGetLong` procedure to convert an input parameter to a `long`. If the input parameter conversion was successful, `VsGetLong` stores the data where `ptr` points and returns `TCL_OK`. If an error occurred during conversion, `VsGetLong` stores nothing where `ptr` points and returns `TCL_ERROR`. The `VsGetLong` procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

ptr A pointer to where the result should go.

The `VsGetShort` Procedure

```
int
VsGetShort(Tcl_Interp *interp,
           String val,
           short *ptr);
```

Use the `VsGetShort` procedure to convert an input parameter to a `short`. If the input parameter conversion was successful, `VsGetShort` stores the data where `ptr` points and returns `TCL_OK`. If an error occurred during conversion, `VsGetShort` stores nothing where `ptr` points and returns `TCL_ERROR`. The `VsGetShort` procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

ptr A pointer to where the result should go.

The `VsGetShortPair` Procedure

```
int
VsGetShortPair(Tcl_Interp *interp,
               String val,
               short *xptr,
               short *yptr);
```

Use the `VsGetShortPair` procedure to convert an input parameter to a `short` pair. If the input parameter conversion was successful, `VsGetShortPair` stores the data where `xptr` and `yptr` point and returns `TCL_OK`. If an error occurred during conversion, `VsGetShortPair` stores nothing where `xptr` and `yptr` point and returns `TCL_ERROR`. The `VsGetShortPair` procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

xptr A pointer to where the first result of the pair should go.

yptr A pointer to where the second result of the pair should go.

The `VsGetString` Procedure

```
int
VsGetString(Tcl_Interp *interp,
            String val,
            String *ptr);
```

Use the `VsGetString` procedure to convert an input parameter to a `String`. If the input parameter conversion was successful, `VsGetString` stores the data where `ptr` points and returns `TCL_OK`. If an error occurred during conversion, `VsGetString` stores nothing where `ptr` points and returns `TCL_ERROR`. The `VsGetString` procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

ptr A pointer to where the result should go.

The `VsGetUnsignedChar` Procedure

```
int
VsGetUnsignedChar(Tcl_Interp *interp,
                  String val,
                  unsigned char *ptr);
```

Use the `VsGetUnsignedChar` procedure to convert an input parameter to an `unsigned char`. If the input parameter conversion was successful, `VsGetUnsignedChar` stores the data where `ptr` points and returns `TCL_OK`. If an error occurred during conversion, `VsGetUnsignedChar` stores nothing where `ptr` points and returns `TCL_ERROR`. The `VsGetUnsignedChar` procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

ptr A pointer to where the result should go.

The `VsGetUnsignedInt` Procedure

```
int
VsGetUnsignedInt(Tcl_Interp *interp,
                 String val,
                 unsigned int *ptr);
```

Use the `VsGetUnsignedInt` procedure to convert an input parameter to an `unsigned int`. If the input parameter conversion was successful, `VsGetUnsignedInt` stores the data where `ptr` points and returns `TCL_OK`. If an error occurred during conversion, `VsGetUnsignedInt` stores nothing where `ptr` points and returns `TCL_ERROR`. The `VsGetUnsignedInt` procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

ptr A pointer to where the result should go.

The `VsGetUnsignedLong` Procedure

```
int
VsGetUnsignedLong(Tcl_Interp *interp,
                  String val,
                  unsigned long *ptr);
```

Use the `VsGetUnsignedLong` procedure to convert an input parameter to an `unsigned long`. If the input parameter conversion was successful, `VsGetUnsignedLong` stores the data where `ptr` points and returns `TCL_OK`. If an error occurred during conversion, `VsGetUnsignedLong` stores nothing where `ptr` points and returns `TCL_ERROR`. The `VsGetUnsignedLong` procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

ptr A pointer to where the result should go.

The `VsGetUnsignedShort` Procedure

```
int
VsGetUnsignedShort(Tcl_Interp *interp,
                   String val,
                   unsigned short *ptr);
```

Use the `VsGetUnsignedShort` procedure to convert an input parameter to an `unsigned short`. If the input parameter conversion was successful, `VsGetUnsignedShort` stores the data where `ptr` points and returns `TCL_OK`. If an error occurred during conversion, `VsGetUnsignedShort` stores nothing where `ptr` points and returns `TCL_ERROR`. The `VsGetUnsignedShort` procedure takes:

interp A pointer to the Tcl interpreter.

val A parameter string to be converted.

ptr A pointer to where the result should go.

C.14 Tcl Command Return Value Generation

The `VsReturnBoolean` Procedure

```
int
VsReturnBoolean(Tcl_Interp *interp,
                Boolean val);
```

Use the `VsReturnBoolean` procedure to convert a return value parameter from a `Boolean`. It puts the string result in the `result` slot of the Tcl interpreter. The `VsReturnBoolean` procedure returns `TCL_OK` and takes:

interp A pointer to the Tcl interpreter.

val A value to be converted.

The `VsReturnInt` Procedure

```
int
VsReturnInt(Tcl_Interp *interp,
            int val);
```

Use the `VsReturnInt` procedure to convert a return value parameter from an `int`. It puts the string result in the `result` slot of the Tcl interpreter. The `VsReturnInt` procedure returns `TCL_OK` and takes:

interp A pointer to the Tcl interpreter.

val A value to be converted.

The `VsReturnIntPair` Procedure

```
int
VsReturnIntPair(Tcl_Interp *interp,
                int x,
                int y);
```

Use the `VsReturnIntPair` procedure to convert a return value parameter from an `int` pair. It puts the string result in the `result` slot of the Tcl interpreter. The `VsReturnIntPair` procedure returns `TCL_OK` and takes:

interp A pointer to the Tcl interpreter.

x A value to be converted to the first of the pair.

y A value to be converted to the second of the pair.

The `VsReturnLong` Procedure

```
int
VsReturnLong(Tcl_Interp *interp,
             long val);
```

Use the `VsReturnLong` procedure to convert a return value parameter from a `long`. It puts the string result in the `result` slot of the Tcl interpreter. The `VsReturnLong` procedure returns `TCL_OK` and takes:

interp A pointer to the Tcl interpreter.

val A value to be converted.

The `VsReturnLongPair` Procedure

```
int
VsReturnLongPair(Tcl_Interp *interp,
                 long x,
                 long y);
```

Use the `VsReturnLongPair` procedure to convert a return value parameter from a `long` pair. It puts the string result in the `result` slot of the Tcl interpreter. The `VsReturnLongPair` procedure returns `TCL_OK` and takes:

interp A pointer to the Tcl interpreter.

x A value to be converted to the first of the pair.

y A value to be converted to the second of the pair.

The `VsReturnFloat` Procedure

```
int
VsReturnFloat(Tcl_Interp *interp,
              float val);
```

Use the `VsReturnFloat` procedure to convert a return value parameter from a `float`. It puts the string result in the `result` slot of the Tcl interpreter. The `VsReturnFloat` procedure returns `TCL_OK` and takes:

interp A pointer to the Tcl interpreter.

val A value to be converted.

The `VsReturnDouble` Procedure

```
int
VsReturnDouble(Tcl_Interp *interp,
               double val);
```

Use the `VsReturnDouble` procedure to convert a return value parameter from a `double`. It puts the string result in the `result` slot of the Tcl interpreter. The `VsReturnDouble` procedure returns `TCL_OK` and takes:

interp A pointer to the Tcl interpreter.

val A value to be converted.

The `VsReturnNull` Procedure

```
int
VsReturnNull(Tcl_Interp *interp);
```

Use the `VsReturnNull` procedure to return a null result. It puts an empty string in the `result` slot of the Tcl interpreter. The `VsReturnNull` procedure returns `TCL_OK` and takes:

interp A pointer to the Tcl interpreter.

The `VsReturnString` Procedure

```
int
VsReturnString(Tcl_Interp *interp,
               String val);
```

Use the `VsReturnString` procedure to convert a return value parameter from a `String`. It puts the string result in the `result` slot of the Tcl interpreter. The `VsReturnString` procedure returns `TCL_OK` and takes:

interp A pointer to the Tcl interpreter.

val A value to be converted.

The `VsReturnStringPair` Procedure

```
int
VsReturnStringPair(Tcl_Interp *interp,
                   String x,
                   String y);
```

Use the `VsReturnStringPair` procedure to convert a return value parameter from a `String` pair. It puts the string result in the `result` slot of the Tcl interpreter. The `VsReturnStringPair` procedure returns `TCL_OK` and takes:

interp A pointer to the Tcl interpreter.

x A value to be converted to the first of the pair.

y A value to be converted to the second of the pair.

The `VsReturnUnsignedLong` Procedure

```
int
VsReturnUnsignedLong(Tcl_Interp *interp,
                     unsigned long val);
```

Use the `VsReturnUnsignedLong` procedure to convert a return value parameter from an `unsigned long`. It puts the string result in the `result` slot of the Tcl interpreter. The `VsReturnUnsignedLong` procedure returns `TCL_OK` and takes:

interp A pointer to the Tcl interpreter.

val A value to be converted.

C.15 Payloads

```
class VsPayload : public VsObj {
  ...
public:
  ...
  int& Channel();
  VsMemBlock& Data();
  VsTimeval& StartingTime();
  VsTimeval& Duration();
  VsTimeval EndingTime();
  ...
};
```

All payloads provide the following parameters useful to modules:

Channel The channel assignment (for multiplexed streams).

Data The block of shared memory where the data for this payload resides.

StartingTime The timestamp for this payload.

Duration The difference between starting time and ending time.

EndingTime The time after the duration.

`VsAudioFragment` Payloads

```
class VsAudioFragment : public VsPayload {
  ...
public:
  VsAudioFragment(const VsTimeval& startingTime, int channel,
                  size_t size, u_short samplesPerSecond,
                  u_char encoding, u_char bitsPerSample,
                  u_char byteOrder, u_char channels);
  u_short& SamplesPerSecond();
  u_char& Encoding();
  u_char& BitsPerSample();
  u_char& ByteOrder();
  u_char& Channels();
  void ComputeDuration();
  ...
};
```

`VsAudioFragment` payloads represent a fragment of audio data. The parameters are:

startingTime The timestamp for this payload.

channel The channel assignment (for multiplexed streams).

size The size of shared memory to allocate for the data.

samplesPerSecond The sample rate.

encoding One of `VsUnknownAudioSampleEncoding`, `VsULawAudioSampleEncoding`, `VsALawAudioSampleEncoding`, `VsLinearAudioSampleEncoding`, or `VsADPCMAudioSampleEncoding`.

bitsPerSample The number of bits in each audio sample.

byteOrder One of `LSBFirst` or `MSBFirst`.

channels The number of audio channels sampled.

`VsCaption` Payloads

```
class VsCaption : public VsPayload {
  ...
public:
  VsCaption(const VsTimeval& startingTime, int channel,
            u_short size, char* rawText);
  char* CaptionText();
  ...
};
```

`VsCaption` payloads represent a line of closed-caption data. The parameters are:

startingTime The timestamp for this payload.

channel The channel assignment (for multiplexed streams).

size The size of shared memory to allocate for the data.

raw Text The raw text.

`VsFinish` Payloads

```
class VsFinish : public VsPayload {
  ...
public:
  VsFinish(const VsTimeval& startingTime, int channel);
  ...
};
```

`VsFinish` payloads mark the end of a temporal sequence of payloads. Modules that have been signaled with the `Stop` (page 176) member function with mode 1 should stop when receiving a `VsFinish` payload. Modules that keep synchronized with the starting times of payloads should resychronize upon receipt of a payload following a `VsFinish` payload. The parameters are:

startingTime The timestamp for this payload.

channel The channel assignment (for multiplexed streams).

`VsFlush` Payloads

```
class VsFlush : public VsPayload {
  ...
public:
  VsFlush(const VsTimeval& startingTime, int channel);
  ...
};
```

`VsFlush` payloads indicate a unexpected discontinuity in a temporal sequence of payloads. Presentation modules waiting for the appropriate time to pass to present data at the right time should stop waiting and flush their data upon receipt of a `VsFlush` payload. The parameters are:

startingTime The timestamp for this payload.

channel The channel assignment (for multiplexed streams).

`VsJpegFrame` Payloads

```
class VsJpegFrame : public VsPayload {
  ...
public:
  VsJpegFrame(const VsTimeval& startingTime, int channel,
              size_t size, const VsTimeval& duration,
              u_short width, u_short height, u_short quality,
              u_char origEncoding);
  u_short& Width();
  u_short& Height();
  u_short& Quality();
  u_char& OrigEncoding();
  ...
};
```

`VsJpegFrame` payloads represent a frame of video data compressed using JPEG compression. The parameters are:

startingTime The timestamp for this payload.

channel The channel assignment (for multiplexed streams).

size The size of shared memory to allocate for the data.

duration The duration of this video frame.

width The width of the image.

height The height of the image.

quality The quality of the compressed image, which ranges from 0 to 100.

origEncoding One of `VsNullVideoPixelEncoding`, `VsGrayVideoPixelEncoding`, `VsColorVideoPixelEncoding`, `VsColorBGRVideoPixelEncoding`, or `VsColorRGBVideoPixelEncoding`.

`VsQRLFrame` Payloads

```
class VsQRLFrame : public VsPayload {
  ...
public:
  VsQRLFrame(const VsTimeval& startingTime, int channel,
             size_t size, const VsTimeval& duration,
             u_char quality,
             u_short width, u_short height, u_short bytesPerLine,
             u_int offset);
  u_char& Quality();
  u_short& Width();
  u_short& Height();
  u_short& BytesPerLine();
  u_int& Offset();
  ...
};
```

`VsQRLFrame` payloads represent a frame of video data compressed using Quantized-Run-Length compression. The parameters are:

startingTime The timestamp for this payload.

channel The channel assignment (for multiplexed streams).

size The size of shared memory to allocate for the data.

duration The duration of this video frame.

quality The quality of the compressed image, which ranges from 0 to 100.

width The width of the image.

height The height of the image.

bytesPerLine The spanning width of the image.

offset The offset of the image.

`VsCCCFrame` Payloads

```
class VsCCCFrame : public VsPayload {
  ...
public:
  VsCCCFrame(const VsTimeval& startingTime, int channel,
             size_t size, const VsTimeval& duration,
             u_short width, u_short height, u_short bytesPerLine);
  u_short& Width();
  u_short& Height();
  u_short& BytesPerLine();
  ...
};
```

`VsCCCFrame` payloads represent a frame of video data compressed using Color-Cell compression. The parameters are:

startingTime The timestamp for this payload.

channel The channel assignment (for multiplexed streams).

size The size of shared memory to allocate for the data.

duration The duration of this video frame.

width The width of the image.

height The height of the image.

bytesPerLine The spanning width of the image.

`VsStart` Payloads

```
class VsStart : public VsPayload {
  ...
public:
  VsStart(const VsTimeval& startingTime, int channel);
  ...
};
```

`VsStart` payloads mark the start of a temporal sequence of payloads. Modules that keep synchronized with the starting times of payloads should resychronize upon receipt of a `VsStart` payload. The parameters are:

startingTime The timestamp for this payload.

channel The channel assignment (for multiplexed streams).

`VsVideoFrame` Payloads

```
class VsVideoFrame : public VsPayload {
  ...
public:
  VsVideoFrame(const VsTimeval& startingTime, int channel,
               size_t size, const VsTimeval& duration,
               u_short width, u_short height, u_short bytesPerLine,
               u_char encoding, u_char depth, u_char bitsPerPixel,
               u_char byteOrder, u_char bitmapUnit,
               u_char bitmapBitOrder, u_char bitmapPad);
  u_short& Width();
  u_short& Height();
  u_short& BytesPerLine();
  u_char& Encoding();
  u_char& Depth();
  u_char& BitsPerPixel();
  u_char& ByteOrder();
  u_char& BitmapUnit();
  u_char& BitmapBitOrder();
  u_char& BitmapPad();
  ...
};
```

`VsVideoFrame` payloads represent a frame of video data. The parameters are:

startingTime The timestamp for this payload.

channel The channel assignment (for multiplexed streams).

size The size of shared memory to allocate for the data.

duration The duration of this video frame.

width The width of the image.

height The height of the image.

bytesPerLine The spanning width of the image.

encoding One of `VsNullVideoPixelEncoding`, `VsGrayVideoPixelEncoding`, `VsColorVideoPixelEncoding`, `VsColorBGRVideoPixelEncoding`, or `VsColorRGBVideoPixelEncoding`.

depth The depth of the image.

bitsPerPixel One of `1`, `4`, `8`, `16`, `24`, or `32`.

byteOrder One of `LSBFirst` or `MSBFirst`.

bitmapUnit One of `8`, `16`, or `32`.

bitmapBitOrder One of `LSBFirst` or `MSBFirst`.

bitmapPad One of `8`, `16`, or `32`.

C.16 Control Panel Procedures

The `VsLabeledPathname` Procedure

```
VsLabeledPathname <parent>.<name> [-label <label>] [-value <value>] \
  [-types <types>] [-mustExist <mustExist>] [-callback <callback>] \
  [-width <width>]
```

The `VsLabeledPathname` procedure creates a control panel entry for displaying and changing a file pathname. Any unrecognized keyword arguments are passed to the first widget that is created. It takes:

parent *(Command Name)* A name of the parent Widget object command.

name *(String)* A child name for the widget.

value *(String)* An initial value for the pathname.

types *(List)* A list to specify which files are to be visible in the file choice dialog box when the user is choosing a new pathname. Each element of the list should be a list of two elements. The first element should be a text string, suitable for inclusion in a menu, describing a set of files. The second element should be a key string for the `regexp` Tcl command that should match pathnames that fit in the set of files.

mustExist *(Boolean)* `1` if the file must exist, `0` otherwise.

callback *(Command String)* A callback command string to be evaluated when the user has chosen a new pathname. The callback string with the new pathname appended is evaluated, and the new pathname is set to the result of the evaluation.

width *(Integer)* A width in pixels to make the text box to hold the pathname.

The `VsLabeledChoice` Procedure

```
VsLabeledChoice <parent>.<name> [-label <label>] [-value <value>] \
  [-choices <choices>] [-callback <callback>]
```

The `VsLabeledChoice` procedure creates a control panel entry for displaying and changing a multiple-choice parameter. Any unrecognized keyword arguments are passed to the first widget that is created. It takes:

parent *(Command Name)* A name of the parent Widget object command.

name *(String)* A child name for the widget.

value *(String)* An initial value for the choice.

choices *(List)* A list specifying the choices. Each element of the list represents a choice. If an element is a list, the first element of the list is taken to be the choice, and the second is used as a representation of the choice in the user interface. If the element is not a list, then the first element is taken to be both the choice and the representation.

callback *(Command String)* A callback command string to be evaluated when the user has chosen a new choice. The callback string with the new choice appended is evaluated, and the result of the evaluation is set to the new choice.

The `VsLabeledScrollbar` Procedure

```
VsLabeledScrollbar <parent>.<name> [-label <label>] [-value <value>] \
  [-callback <callback>] [-converter <converter>] [-inverter <inverter>] \
  [-continuous <continuous>] [-width <width>] [-valueWidth <valueWidth>]
```

The `VsLabeledScrollbar` procedure creates a control panel entry for displaying and changing a numeric parameter that can vary within a range. Any unrecognized keyword arguments are passed to the first widget that is created. It takes:

parent *(Command Name)* A name of the parent Widget object command.

name *(String)* A child name for the widget.

value *(String)* An initial value for the scrollbar.

callback *(Command String)* A callback command string to be evaluated when the user has chosen a new value with the scrollbar. The callback string with the new converted value appended is evaluated, and the result of the evaluation is inverted and set to the new scrollbar.

converter *(Command String)* A command string to be evaluated when a value in scroll units needs to be converted to module units. Scroll units are always real numbers with minimum 0 and maximum 1. Module units are defined by the module option subcommand. Useful procedures here are `vsLinearConverter` (page 197) and `vsRoundingLinearConverter` (page 198).

inverter *(Command String)* A command string to be evaluated when a value in module units needs to be converted back to scroll units. Scroll units are always real numbers with minimum 0 and maximum 1. Module units are defined by the module option subcommand. A useful procedure is `vsLinearInverter` (page 198).

continuous *(Boolean)* `1` to call the converter, callback, and inverter with each movement of the mouse while the button is held down on the scrollbar. `0` to call the converter, callback, and inverter only when the mouse button is released.

width *(Integer)* A width in pixels to make the scrollbar.

valueWidth *(Integer)* A width in pixels to make the text box to hold the value.

The `vsLinearConverter` Procedure

```
vsLinearConverter <min> <max> <scrollVal>
  ==> <moduleVal>
```

The `vsLInearConverter` procedure converts `VsLabeledScrollbar` (page 197) scroll values to module values for module values that vary linearly and continuously in a range. It takes:

min *(Double)* The minimum value in module units.

max *(Double)* The maximum value in module units.

scrollVal *(Double)* The input value, in scroll units, to be converted to module units.

It returns:

moduleVal *(Double)* The output value, in module units, converted from scroll units.

The `vsRoundingLinearConverter` Procedure

```
vsRoundingLinearConverter <min> <max> <scrollVal>
  ==> <moduleVal>
```

The `vsRoundingLinearConverter` procedure converts `VsLabeledScrollbar` (page 197) scroll values to module values for module values that vary linearly and discretely in a range. It takes:

min *(Double)* The minimum value in module units.

max *(Double)* The maximum value in module units.

scrollVal *(Double)* The input value, in scroll units, to be converted to module units.

It returns:

moduleVal *(Integer)* The output value, in module units, converted from scroll units.

The `vsLinearInverter` Procedure

```
vsLinearInverter <min> <max> <moduleVal>
  ==> <scrollVal>
```

The `vsLInearInverter` procedure inverts module values to `VsLabeledScrollbar` (page 197) scroll values for module values that vary linearly in a range. It takes:

min *(Double)* The minimum value in module units.

max *(Double)* The maximum value in module units.

moduleVal *(Double)* The input value, in module units, to be inverted to scroll units.

It returns:

scrollVal *(Double)* The output value, in scroll units, inverted from module units.

Appendix D

Tcl Support For A Graphical User Interface

D.1 Object Types

Here is a summary of the object commands provided to support a graphical user interface under the X Window System using the Xt and Xaw toolkits. The complete object type hierarchy is shown in Figure D.1.

The `xt` Object Command

The `xt` object command provides the top-level interface to the tclXt and tclXaw libraries. It is used for initialization of the application user interface. The most common use of the `xt` object command is the `appInitialize` (page 201) Xt subcommand. Xt subcommands are described in Section D.2.

XtAppContext Object Commands

XtAppContext object commands provide an interface to XtAppContext data structures and functions provided by the Xt library. They are used to access the global application state. XtAppContext object commands are most commonly created by the `appInitialize` (page 201) Xt command, but can also be created with the `createApplicationContext` (page 201) Xt command. XtAppContext subcommands are described in Section D.3.

Display Object Commands

Display object commands provide an interface to Display data structures and functions provided by the Xt library. They are used to access the state relating to the connection from the application to the X server. Display object commands are created by the `display` (page 217) Object subcommand. Display subcommands are described in Section D.4.

XEvent Object Commands

XEvent object commands provide an interface to XEvent data structures. They are used to access event parameters such as cursor position, and application components such as windows and menus. The `%event` XEvent object command represents the current event during the evaluation of callback and translation commands strings. XEvent subcommands are described in Section D.5.

```
Display
Object
    Rect
        Sme
            SmeBSB
            SmeLine
        UnNamedObj
            Core
                Composite
                    Box
                    Constraint
                        Form
                            Dialog, Viewport
                        Paned, Tree
                        WorkSpace
                            Graph
                    Porthole
                    Shell
                        OverrideShell
                            SimpleMenu
                        WMShell
                            VendorShell
                                TopLevelShell
                                    ApplicationShell
                                TransientShell
                Simple
                    Clock, Grip
                    Label
                        Command
                            MenuButton, Repeater, Toggle
                    List, Logo, Mailbox, Panner, Scrollbar, StripChart
                    Text
                        AsciiText
    TextSink
        AsciiSink
    TextSrc
        AsciiSrc
ObjectClass
    RectClass
        SmeClass
            SmeBSBClass, SmeLineClass
    UnNamedObjClass
        CoreClass
            CompositeClass
                BoxClass
                ConstraintClass
                    FormClass
                        DialogClass, ViewportClass
                    PanedClass, TreeClass
                    WorkSpaceClass
                        GraphClass
                PortholeClass
                ShellClass
                    OverrideShellClass
                        SimpleMenuClass
                    WMShellClass
                        VendorShellClass
                            TopLevelShellClass
                                ApplicationShellClass
                            TransientShellClass
            SimpleClass
                ClockClass, GripClass
                LabelClass
                    CommandClass
                        MenuButtonClass, RepeaterClass, ToggleClass
                ListClass, LogoClass, MailboxClass, PannerClass
                ScrollbarClass, StripChartClass
                TextClass
                    AsciiTextClass
    TextSinkClass
        AsciiSinkClass
    TextSrcClass
        AsciiSrcClass
XEvent, Xt, XtAppContext
```

Figure D.1: The Complete List Of GUI Object Types.

Widget Object Commands

Widget object commands provide an interface to Widget data structures and procedures provided by the Xt and Xaw library. They are used to manage application components such as windows and menus. Widget object commands are created by the appropriate WidgetClass (page 201) object commands.

Various Widget subcommands are described in Sections D.11, D.12, D.13, D.14, D.15, D.16, D.17, D.18, D.19, D.20, D.21, D.22, D.23, D.24, D.25, D.26, and D.27.

WidgetClass Object Commands

WidgetClass object commands provide an interface to WidgetClass data structures and procedures provided by the Xt and Xaw library. They are used to create widgets and manage widgets. WidgetClass object commands are created by the `appInitalize` (page 201) or `toolkitInitialize` (page 202) Xt commands.

The WidgetClass object commands are `Sme`, `SmeBSB`, `SmeLine`, `UnNamedObj`, `Core`, `Composite`, `Box`, `Constraint`, `Form`, `Dialog`, `Viewport`, `Paned`, `Tree`, `WorkSpace`, `Graph`, `Porthole`, `Shell`, `OverrideShell`, `SimpleMenu`, `WMShell`, `VendorShell`, `TopLevelShell`, `ApplicationShell`, `TransientShell`, `Simple`, `Clock`, `Grip`, `Label`, `Command`, `MenuButton`, `Repeater`, `Toggle`, `List`, `Logo`, `Mailbox`, `Panner`, `Scrollbar`, `StripChart`, `Text`, `AsciiText`, `TextSink`, `AsciiSink`, `TextSrc`, and `AsciiSrc`.

WidgetClass subcommands are described in Sections D.6, D.7, D.8, and D.9.

D.2 Xt Subcommands

The `appInitialize` Xt Subcommand

```
xt appInitialize <appContextName> <class> <argvVarname> \
    <fallbackResources>
```

The `appInitialize` Xt subcommand provides an interface to the `XtAppInitialize` function. It takes:

appContextName *(String)* A name for the XtAppContext object command to be created.

class *(String)* An application class.

argvVarname *(Variable Name)* A name of a variable where the command-line arguments to this program are stored. This variable is updated, with command-line options removed as they are recognized and processed.

fallbackResources *(List)* A list of application fallback resource specifications.

The equivalent C code for this subcommand would be:

```
XtAppInitialize(&appContext, applicationClass, options,
                numOptions, &argcInOut, argvInOut,
                fallbackResources, args, numArgs);
```

The `createApplicationContext` Xt Subcommand

```
xt createApplicationContext <name>
```

The `createApplicationContext` Xt subsubcommand provides an interface to the `XtCreateApplicationContext` function. It takes:

name *(String)* A name for the XtAppContext object command to be created.

The equivalent C code for this subcommand would be:

```
XtCreateApplicationContext();
```

The findFile Xt Subcommand

```
xt findFile <path> <substitutions> <subcommand>
  ==> <filename>
```

The `findFile` Xt subcommand provides an interface to the `XtFindFile` function. It takes:

- **path** *(String)* A file search path.
- **substitutions** *(Substitution List)* A substitution list.
- **command** *(Command String)* A command string to be evaluated for each file found. The command is evaluated in a procedure body with the following local variables bound:
  - **filename** *(String)* The name of the file.

  The result of the evaluation should be 1 if the file name is acceptable, 0 otherwise.

It returns:

- **filename** *(String)* The file name that was accepted by the subcommand string.

The equivalent C code for this subcommand would be:

```
filename = XtFindFile(path, subs, numSubs, predicate);
```

The toolkitInitialize Xt Subcommand

```
xt toolkitInitialize
```

The `toolkitInitialize` Xt subcommand provides an interface to the `XtToolkitInitialize` function. The equivalent C code for this subcommand would be:

```
XtToolkitInitialize();
```

D.3 XtAppContext Subcommands

The addActionHook XtAppContext Subcommand

```
<appContext> addActionHook <subcommand> ==> <id>
```

The `addActionHook` XtXtAppContext Subcommand provides an interface to the `XtAppAddActionHook` function. It takes:

- **command** *(Command String)* A command string to be evaluated when the action occurs. The command is evaluated in a procedure body with the following local variables bound:
  - **widget** *(Command Name)* The name of the widget.
  - **action_name** *(String)* The name of the action.
  - **event** *(Command Name)* The name of the event.
  - **params** *(List)* The action parameters.

It returns:

- **id** *(Integer)* The handle to be used with the `removeActionHook` XtAppContext subcommand.

The equivalent C code for this subcommand would be:

```
id = XtAppAddActionHook(appContext, proc, closure);
```

The removeActionHook XtAppContext Subcommand

```
<appContext> removeActionHook <id>
```

The **removeActionHook** XtAppContext subcommand provides an interface to the **XtRemoveActionHook** function. It takes:

- **id** *(Integer)* A handle returned from the **addActionHook** XtAppContext subcommand.

The equivalent C code for this subcommand would be:

```
XtRemoveActionHook(id);
```

The addInput XtAppContext Subcommand

```
<appContext> addInput <source> <condition> <subcommand>
  ==> <id>
```

The **addInput** XtAppContext subcommand provides an interface to the **XtAppAddInput** function. It takes:

- **source** *(Integer)* A file descriptor open for input.
- **condition** *(Integer)* A condition indicating when the file is ready.
- **command** *(Command String)* A command string to be evaluated when the file is ready. The command is evaluated in a procedure body with the following local variables bound:
  - **source** *(Integer)* The file descriptor.
  - **id** *(Integer)* The handle to be used with the **removeInput** XtAppContext subcommand.

It returns:

- **id** *(Integer)* The handle to be used with the **removeInput** XtAppContext subcommand.

The equivalent C code for this subcommand would be:

```
id = XtAppAddInput(appContext, source, condition, proc, closure);
```

The removeInput XtAppContext Subcommand

```
<appContext> removeInput <id>
```

The **removeInput** XtAppContext subcommand provides an interface to the **XtRemoveInput** function. It takes:

- **id** *(Integer)* A handle returned from the **addInput** XtAppContext subcommand.

The equivalent C code for this subcommand would be:

```
XtRemoveInput(id);
```

The `addtimeout` XtAppContext Subcommand

```
<appContext> addtimeout <interval> <subcommand>
  ==> <id>
```

The `addtimeout` XtAppContext subcommand provides an interface to the `XtAppAddtimeout` function. It takes:

interval *(Integer)* A number of milliseconds to pass before evaluating the subcommand.

command *(Command String)* A command string to be evaluated after the time has elapsed. The command is evaluated in a procedure body with the following local variables bound:

> **id** *(Integer)* The handle to be used with the `removetimeout` XtAppContext subcommand.

It returns:

id *(Integer)* The handle to be used with the `removetimeout` XtAppContext subcommand.

The equivalent C code for this subcommand would be:

```
id = XtAppAddtimeout(appContext, interval, proc, closure);
```

The `removetimeout` XtAppContext Subcommand

```
<appContext> removetimeout <id>
```

The `removetimeout` XtAppContext subcommand provides an interface to the `XtRemovetimeout` function. It takes:

id *(Integer)* A handle returned from the `addtimeout` XtAppContext subcommand.

The equivalent C code for this subcommand would be:

```
XtRemovetimeout(timer);
```

The `addWorkProc` XtAppContext Subcommand

```
<appContext> addWorkProc <subcommand>
  ==> <id>
```

The `addWorkProc` XtAppContext subcommand provides an interface to the `XtAppAddWorkProc` function. It takes:

command *(Command String)* A command string to be evaluated. The command is evaluated in a procedure body. The result of the evaluation should be 1 if it is done, and 0 if it should be called again.

It returns:

id *(Integer)* The handle to be used with the `removeWorkProc` XtAppContext subcommand.

The equivalent C code for this subcommand would be:

```
id = XtAppAddWorkProc(appContext, proc, closure);
```

The `removeWorkProc` XtAppContext Subcommand

```
<appContext> removeWorkProc <id>
```

The `removeWorkProc` XtAppContext subcommand provides an interface to the `XtRemoveWorkProc` function. It takes:

id *(Integer)* A handle returned from the `addWorkProc` XtAppContext subcommand.

The equivalent C code for this subcommand would be:

```
XtRemoveWorkProc(id);
```

The `destroy` XtAppContext Subcommand

```
<appContext> destroy
```

The `destroy` XtAppContext subcommand provides an interface to the `XtDestroyApplicationContext` function. The equivalent C code for this subcommand would be:

```
XtDestroyApplicationContext(appContext);
```

The `error` XtAppContext Subcommand

```
<appContext> error <message>
```

The `error` XtAppContext subcommand provides an interface to the `XtAppError` function. It takes:

message *(String)* An error message text.

The equivalent C code for this subcommand would be:

```
XtAppError(appContext, message);
```

The `errorMsg` XtAppContext Subcommand

```
<appContext> errorMsg <name> <type> <class> <default> [<param>]...
```

The `errorMsg` XtAppContext subcommand provides an interface to the `XtAppErrorMsg` function. It takes:

name *(String)* A name of an error message.

type *(String)* A type of an error message.

class *(String)* A class of an error message.

default *(String)* A default error message text.

param *(String)* A parameter to the error message.

The equivalent C code for this subcommand would be:

```
XtAppErrorMsg(appContext, name, type, class, dflt,
              params, &numParams);
```

The `getSelectiontimeout` XtAppContext Subcommand

```
<appContext> getSelectiontimeout
  ==> <timeout>
```

The `getSelectiontimeout` XtAppContext subcommand provides an interface to the `XtAppGetSelectiontimeout` function. It returns:

> **timeout** *(Integer)* The selection timeout.

The equivalent C code for this subcommand would be:

```
timeout = XtAppGetSelectiontimeout(appContext);
```

The `mainLoop` XtAppContext Subcommand

```
<appContext> mainLoop
```

The `mainLoop` XtAppContext subcommand provides an interface to the `XtAppMainLoop` function. The equivalent C code for this subcommand would be:

```
XtAppMainLoop(appContext);
```

The `nextEvent` XtAppContext Subcommand

```
<appContext> nextEvent <name>
```

The `nextEvent` XtAppContext subcommand provides an interface to the `XtAppNextEvent` function. It takes:

> **name** *(String)* A name for the event object command to be created.

The equivalent C code for this subcommand would be:

```
XtAppNextEvent(appContext, event);
```

The `openDisplay` XtAppContext Subcommand

```
<appContext> openDisplay <name> <displayString> <applicationName> \
    <applicationClass> <argvVarname>
```

The `openDisplay` XtAppContext subcommand provides an interface to the `XtOpenDisplay` function. It takes:

> **name** *(String)* A name for the Display object command to be created.
>
> **displayString** *(String)* An X display string.
>
> **applicationName** *(String)* An application name.
>
> **applicationClass** *(String)* An application class.
>
> **argvVarname** *(Variable Name)* A name of a variable where the subcommand-line arguments to this program are stored. This variable is updated, with subcommand-line options removed as they are recognized and processed.

The equivalent C code for this subcommand would be:

```
XtOpenDisplay(appContext, dpyString, applicationName,
              applicationClass, options, numOptions,
              &argcInOut, argvInOut);
```

The `peekEvent` XtAppContext Subcommand

```
<appContext> peekEvent <event>
  ==> <present>
```

The `peekEvent` XtAppContext subcommand provides an interface to the `XtAppPeekEvent` function. It takes:

event *(String)* A type of an event.

It returns:

present *(Boolean)* 1 if an event is present, 0 otherwise.

The equivalent C code for this subcommand would be:

```
present = XtAppPeekEvent(appContext, event);
```

The `pending` XtAppContext Subcommand

```
<appContext> pending
  ==> <pending>
```

The `pending` XtAppContext subcommand provides an interface to the `XtAppPending` function. It returns:

pending *(Integer)* 1 if an event is pending, 0 otherwise.

The equivalent C code for this subcommand would be:

```
pending = XtAppPending(appContext);
```

The `processEvent` XtAppContext Subcommand

```
<appContext> processEvent <mask>
```

The `processEvent` XtAppContext subcommand provides an interface to the `XtAppProcessEvent` function. It takes:

mask *(Integer)* An input mask.

The equivalent C code for this subcommand would be:

```
XtAppProcessEvent(appContext, mask);
```

The `setSelectiontimeout` XtAppContext Subcommand

```
<appContext> setSelectiontimeout <milliseconds>
```

The `setSelectiontimeout` XtAppContext subcommand provides an interface to the `XtAppSetSelectiontimeout` function. It takes:

milliseconds *(Integer)* A number of milliseconds to set the selection timeout.

The equivalent C code for this subcommand would be:

```
XtAppSetSelectiontimeout(appContext, milliseconds);
```

The `setErrorHandler` XtAppContext Subcommand

```
<appContext> setErrorHandler <subcommand>
```

The `setErrorHandler` XtAppContext subcommand provides an interface to the `XtAppSetErrorHandler` function. It takes:

command *(Command String)* A command string to be evaluated when an error is being reported. The command is evaluated in a procedure body with the following local variables bound:

> **msg** *(String)* The error message text.

The equivalent C code for this subcommand would be:

```
XtAppSetErrorHandler(appContext, handler);
```

The `setErrorMsgHandler` XtAppContext Subcommand

```
<appContext> setErrorMsgHandler <subcommand>
```

The `setErrorMsgHandler` XtAppContext subcommand provides an interface to the `XtAppSetErrorMsgHandler` function. It takes:

command *(Command String)* A command string to be evaluated when an error message is being reported. The command is evaluated in a procedure body with the following local variables bound:

> **name** *(String)* The name of the error message.
> **type** *(String)* The type of the error message.
> **class** *(String)* The class of the error message.
> **default** *(String)* The default error message text.
> **params** *(String)* A list of parameters to the error message.

The equivalent C code for this subcommand would be:

```
XtAppSetErrorMsgHandler(appContext, handler);
```

The `setFallbackResources` XtAppContext Subcommand

```
<appContext> setFallbackResources <fallbackResources>
```

The `setFallbackResources` XtAppContext subcommand provides an interface to the `XtAppSetFallbackResources` function. It takes:

fallbackResources *(List)* A list of application fallback resource specifications.

The equivalent C code for this subcommand would be:

```
XtAppSetFallbackResources(appContext, frl);
```

The `setWarningHandler` XtAppContext Subcommand

```
<appContext> setWarningHandler <subcommand>
```

The `setWarningHandler` XtAppContext subcommand provides an interface to the `XtAppSetWarningHandler` function. It takes:

> **command** *(Command String)* A command string to be evaluated when a warning is being reported. The command is evaluated in a procedure body with the following local variables bound:
>
> > **msg** *(String)* The warning message text.

The equivalent C code for this subcommand would be:

```
XtAppSetWarningHandler(appContext, handler);
```

The `setWarningMsgHandler` XtAppContext Subcommand

```
<appContext> setWarningMsgHandler <subcommand>
```

The `setWarningMsgHandler` XtAppContext subcommand provides an interface to the `XtAppSetWarningMsgHandler` function. It takes:

> **command** *(Command String)* A command string to be evaluated when a warning message is being reported. The command is evaluated in a procedure body with the following local variables bound:
>
> > **name** *(String)* The name of the warning message.
> > **type** *(String)* The type of the warning message.
> > **class** *(String)* The class of the warning message.
> > **default** *(String)* The default warning message text.
> > **params** *(String)* A list of parameters to the warning message.

The equivalent C code for this subcommand would be:

```
XtAppSetWarningMsgHandler(appContext, handler);
```

The `warning` XtAppContext Subcommand

```
<appContext> warning <message>
```

The `warning` XtAppContext subcommand provides an interface to the `XtAppWarning` function. It takes:

> **message** *(String)* A warning message text.

The equivalent C code for this subcommand would be:

```
XtAppWarning(appContext, message);
```

The `warningMsg` XtAppContext Subcommand

```
<appContext> warningMsg <name> <type> <class> <default> \
    [<param>]...
```

The `warningMsg` XtAppContext subcommand provides an interface to the `XtAppWarningMsg` function. It takes:

name *(String)* A name of a warning message.

type *(String)* A type of a warning message.

class *(String)* A class of a warning message.

default *(String)* A default warning message text.

param *(String)* A parameter to the warning message.

The equivalent C code for this subcommand would be:

```
XtAppWarningMsg(appContext, name, type, class, dflt,
                params, &numParams);
```

The `addGlobalActions` XtAppContext Subcommand

```
<appContext> addGlobalActions
```

The `addGlobalActions` XtAppContext subcommand provides an interface to the `XawSimpleMenuAddGlobalActions` function. The equivalent C code for this subcommand would be:

```
XawSimpleMenuAddGlobalActions(appCon);
```

D.4 Display Subcommands

The `createShell` Display Subcommand

```
<display> createShell <applicationName> <applicationClass> \
    <widgetClass>
```

The `createShell` Display subcommand provides an interface to the `XtAppCreateShell` function. It takes:

applicationName *(String)* An application name.

applicationClass *(String)* An application class.

widgetClass *(String)* A name for the shell Widget object command to be created.

The equivalent C code for this subcommand would be:

```
XtAppCreateShell(applicationName, applicationClass,
                 widgetClass, dpy, args, numArgs);
```

The `applicationContext` Display Subcommand

```
<display> applicationContext [<name>]
  ==> <appContext>
```

The `applicationContext` Display subcommand provides an interface to the `XtDisplayToApplicationContext` function. It takes:

name *(String)* A name for the XtAppContext object command to be created, if an object command for the XtAppContext does not yet exist.

It returns:

appContext *(Command Name)* The name of the XtAppContext object command.

The equivalent C code for this subcommand would be:

```
appContext = XtDisplayToApplicationContext(dpy);
```

The `close` Display Subcommand

```
<display> close
```

The `close` Display subcommand provides an interface to the `XtCloseDisplay` function. The equivalent C code for this subcommand would be:

```
XtCloseDisplay(dpy);
```

The `getApplicationNameAndClass` Display Subcommand

```
<display> getApplicationNameAndClass
  ==> <name> <class>
```

The `getApplicationNameAndClass` Display subcommand provides an interface to the `XtGetApplicationNameAndClass` function. It returns:

name *(String)* The application name.

class *(String)* The application class.

The equivalent C code for this subcommand would be:

```
XtGetApplicationNameAndClass(dpy, &name, &class);
```

The `getMultiClickTime` Display Subcommand

```
<display> getMultiClickTime
  ==> <time>
```

The `getMultiClickTime` Display subcommand provides an interface to the `XtGetMultiClickTime` function. It returns:

time *(Integer)* The multi-click time in milliseconds.

The equivalent C code for this subcommand would be:

```
time = XtGetMultiClickTime(dpy);
```

The `lastTimestampProcessed` Display Subcommand

```
<display> lastTimestampProcessed
  ==> <last>
```

The `lastTimestampProcessed` Display subcommand provides an interface to the `XtLastTimestampProcessed` function. It returns:

last *(Integer)* The last timestamp processed.

The equivalent C code for this subcommand would be:

```
last = XtLastTimestampProcessed(dpy);
```

The `resolvePathname` Display Subcommand

```
<display> resolvePathname <type> <filename> <suffix> <path> \
    <substitutions> <subcommand>
  ==> <resolved>
```

The `resolvePathname` Display subcommand provides an interface to the `XtResolvePathname` function. It takes:

type *(String)* A file type.

filename *(String)* A file name.

suffix *(String)* A file suffix.

path *(String)* A file search path.

substitutions *(Substitution List)* A file substitution list.

command *(Command String)* A command string to be evaluated for each file found. The command is evaluated in a procedure body with the following local variables bound:

> **filename** *(String)* The name of the file.

The result of the evaluation should be 1 if the file name is acceptable, 0 otherwise.

It returns:

resolved *(String)* The file name that was accepted by the subcommand string.

The equivalent C code for this subcommand would be:

```
resolved = XtResolvePathname(dpy, type, filename, suffix, path,
                             subs, numSubs, predicate);
```

The `setMultiClickTime` Display Subcommand

```
<display> setMultiClickTime <milliseconds>
```

The `setMultiClickTime` Display subcommand provides an interface to the `XtSetMultiClickTime` function. It takes:

milliseconds *(Integer)* A number of milliseconds to set the multi-click time.

The equivalent C code for this subcommand would be:

```
XtSetMultiClickTime(dpy, milliseconds);
```

The `windowToWidget` Display Subcommand

```
<display> windowToWidget <window>
  ==> <widget>
```

The `windowToWidget` Display subcommand provides an interface to the `XtWindowToWidget` function. It takes:

> **window** *(Integer)* A window id.

It returns:

> **widget** *(Command Name)* A Widget object command name.

The equivalent C code for this subcommand would be:

```
widget = XtWindowToWidget(dpy, window);
```

D.5 XEvent Subcommands

The `destroy` XEvent Subcommand

```
<event> destroy
```

The `destroy` XEvent subcommand provides an interface to the `XtFree` function. The equivalent C code for this subcommand would be:

```
XtFree((char*)event);
```

The `dispatch` XEvent Subcommand

```
<event> dispatch
  ==> <dispatched>
```

The `dispatch` XEvent subcommand provides an interface to the `XtDispatchEvent` function. It returns:

> **dispatched** *(Boolean)* 1 if the event was dispatched, 0 otherwise.

The equivalent C code for this subcommand would be:

```
dispatched = XtDispatchEvent(event);
```

The `position` XEvent Subcommand

```
<event> position
  ==> <x> <y>
```

The `position` XEvent subcommand provides access to the cursor position of an event. It returns:

> **x** *(Integer)* The x coorrdinate.
>
> **y** *(Integer)* The y coordinate.

The equivalent C code for this subcommand would be:

```
x, y = position of event
```

D.6 ObjectClass Subcommands

The `createManagedWidget` ObjectClass Subcommand

```
<objectClass> createManagedWidget <name> <parent> \
    [<resource> <value>]...
```

The `createManagedWidget` ObjectClass subcommand provides an interface to the `XtCreateManagedWidget` function. It takes:

name *(String)* A child name for the widget.

parent *(Command Name)* A name of the parent Widget object command.

resource *(String)* A resource name. If it starts with a hyphen, the hypen is removed.

value *(String)* A resource value.

The equivalent C code for this subcommand would be:

```
XtCreateManagedWidget(name, objectClass, parent, args, numArgs);
```

The `createWidget` ObjectClass Subcommand

```
<objectClass> createWidget <name> <parent> [<resource> <value>]...
```

The `createWidget` ObjectClass subcommand provides an interface to the `XtCreateWidget` function. It takes:

name *(String)* A child name for the widget.

parent *(Command Name)* A name of the parent Widget object command.

resource *(String)* A resource name. If it starts with a hyphen, the hypen is removed.

value *(String)* A resource value.

The equivalent C code for this subcommand would be:

```
XtCreateWidget(name, objectClass, parent, args, numArgs);
```

The `getConstraintResourceList` ObjectClass Subcommand

```
<objectClass> getConstraintResourceList
  ==> <resources>
```

The `getConstraintResourceList` ObjectClass subcommand provides an interface to the `XtGetConstraintResourceList` function. It returns:

resources *(List)* A list of resource information. Each element of the list is a list of resource name, resource class, and resource type.

The equivalent C code for this subcommand would be:

```
XtGetConstraintResourceList(widgetClass, &res, &numRes);
```

The getResourceList ObjectClass Subcommand

```
<objectClass> getResourceList
  ==> <resources>
```

The `getResourceList` ObjectClass subcommand provides an interface to the `XtGetResourceList` function. It returns:

resources *(List)* A list of resource information. Each element of the list is a list of resource name, resource class, and resource type.

The equivalent C code for this subcommand would be:

```
XtGetResourceList(widgetClass, &res, &numRes);
```

D.7 RectClass Subcommands

The RectClass Instance Creation Subcommand

```
<rectClass> <parent>.<name> [<resource> <value>]...
```

The RectClass Instance Creation subcommand provides an interface to the `XtCreateManagedWidget` function. It takes:

parent *(Command Name)* A name of the parent Widget object command.

name *(String)* A child name for the widget.

resource *(String)* A resource name. If it starts with a hyphen, the hypen is removed.

value *(String)* A resource value.

The equivalent C code for this subcommand would be:

```
XtCreateManagedWidget(name, objectClass, parent, args, numArgs);
```

D.8 WidgetClass Subcommands

The getActionList WidgetClass Subcommand

```
<widgetClass> getActionList
  ==> <actions>
```

The `getActionList` WidgetClass subcommand provides an interface to the `XtGetActionList` function. It returns:

actions *(List)* The list of actions.

The equivalent C code for this subcommand would be:

```
XtGetActionList(widgetClass, &actions, &numActions);
```

D.9 ShellClass Subcommands

The ShellClass Instance Creation Subcommand

```
<shellClass> <parent>.<name> [<resource> <value>]...
```

The ShellClass Instance Creation subcommand provides an interface to the `XtCreatePopupShell` function. It takes:

parent *(Command Name)* A name of the parent Widget object command.

name *(String)* A child name for the widget.

resource *(String)* A resource name. If it starts with a hyphen, the hypen is removed.

value *(String)* A resource value.

The equivalent C code for this subcommand would be:

```
XtCreatePopupShell(name, objectClass, parent, args, numArgs);
```

The `createPopupShell` ShellClass Subcommand

```
<shellClass> createPopupShell <name> <parent> \
    [<resource> <value>]...
```

The `createPopupShell` ShellClass subcommand provides an interface to the `XtCreatePopupShell` function. It takes:

name *(String)* A child name for the widget.

parent *(Command Name)* A name of the parent Widget object command.

resource *(String)* A resource name. If it starts with a hyphen, the hypen is removed.

value *(String)* A resource value.

The equivalent C code for this subcommand would be:

```
XtCreatePopupShell(name, objectClass, parent, args, numArgs);
```

D.10 Object Commands

The `checkSubclassFlag` Object Subcommand

```
<object> checkSubclassFlag <typeFlag>
  ==> <isSubclass>
```

The `checkSubclassFlag` Object subcommand provides an interface to the `Xtchecksubclassflag` function. It takes:

typeFlag *(Integer)* A type flag.

It returns:

isSubclass *(Boolean)*

The equivalent C code for this subcommand would be:

```
isSubclass = Xtchecksubclassflag(widget, typeFlag);
```

The `class` Object Subcommand

```
<object> class
  ==> <class>
```

The `class` Object subcommand provides an interface to the `XtClass` function. It returns:

class *(Command Name)* The ObjectClass object command.

The equivalent C code for this subcommand would be:

```
class = XtClass(object);
```

The `destroy` Object Subcommand

```
<object> destroy
```

The `destroy` Object subcommand provides an interface to the `XtDestroyWidget` function. The equivalent C code for this subcommand would be:

```
XtDestroyWidget(widget);
```

The `display` Object Subcommand

```
<object> display [<name>]
  ==> <display>
```

The `display` Object subcommand provides an interface to the `XtDisplayOfObject` function. It takes:

name *(String)* A name for the Display object command to be created, if an object command for the display does not yet exist.

It returns:

display *(Command Name)* The name of the Display object command.

The equivalent C code for this subcommand would be:

```
display = XtDisplayOfObject(widget);
```

The `getValues` Object Subcommand

```
<object> getValues <resource> [<resource>]...
  ==> <value> [<value>]...
```

The `getValues` Object subcommand provides an interface to the `XtGetValues` function. It takes:

resource *(String)* A resource name. If it starts with a hyphen, the hypen is removed.

It returns:

value *(String)* The value of the resource.

The equivalent C code for this subcommand would be:

```
XtGetValues(widget, args, numArgs);
```

The `isObject` Object Subcommand

```
<object> isObject
  ==> <isObject>
```

The `isObject` Object subcommand provides an interface to the `XtIsObject` function. It returns:

> **isObject** *(Boolean)* 1 if it is an object, 0 otherwise.

The equivalent C code for this subcommand would be:

```
isObject = XtIsObject(object);
```

The `isSubclassOf` Object Subcommand

```
<object> isSubclassOf <widgetClass> <flagClass> <typeFlag>
  ==> <isSubclass>
```

The `isSubclassOf` Object subcommand provides an interface to the `Xtissubclassof` function. It takes:

> **widgetClass** *(Command Name)* A name of the WidgetClass object command.
>
> **flagClass** *(Command Name)* A name of the flag WidgetClass object command.
>
> **typeFlag** *(Integer)* A type flag.

It returns:

> **isSubclass** *(Boolean)* 1 if the widget class is a subclass of the flag class, 0 otherwise. Or vice-versa. Whatever.

The equivalent C code for this subcommand would be:

```
isSubclass = Xtissubclassof(object, widgetClass, flagClass,
                            typeFlag);
```

The `name` Object Subcommand

```
<object> name
  ==> <name>
```

The `name` Object subcommand provides an interface to the `XtName` function. It returns:

> **name** *(String)* The child name of the object.

The equivalent C code for this subcommand would be:

```
name = XtName(object);
```

The `setValues` Object Subcommand

```
<object> setValues <resource> <value> [<resource> <value>]...
```

The `setValues` Object subcommand provides an interface to the `XtSetValues` function. It takes:

resource *(String)* A resource name. If it starts with a hyphen, the hypen is removed.

value *(String)* A resource value

The equivalent C code for this subcommand would be:

```
XtSetValues(widget, args, numArgs);
```

The `superclass` Object Subcommand

```
<object> superclass
  ==> <superClass>
```

The `superclass` Object subcommand provides an interface to the `XtSuperclass` function. It returns:

superClass *(Command Name)* The superclass object command name.

The equivalent C code for this subcommand would be:

```
superClass = XtSuperclass(object);
```

The `window` Object Subcommand

```
<object> window
  ==> <window>
```

The `window` Object subcommand provides an interface to the `XtWindowOfObject` function. It returns:

window *(Integer)* The window id.

The equivalent C code for this subcommand would be:

```
window = XtWindowOfObject(widget);
```

D.11 Rect Subcommands

The `isManaged` Rect Subcommand

```
<rect> isManaged
  ==> <isManaged>
```

The `isManaged` Rect subcommand provides an interface to the `XtIsManaged` function. It returns:

isManaged *(Boolean)* 1 if the widget is managed, 0 otherwise.

The equivalent C code for this subcommand would be:

```
isManaged = XtIsManaged(rectobj);
```

D.12 Widget Subcommands

The `applicationContext` Widget Subcommand

```
<widget> applicationContext [<name>]
  ==> <appContext>
```

The `applicationContext` Widget subcommand provides an interface to the `XtWidgetToApplicationContext` function. It takes:

name *(Command Name)* A name for the XtAppContext object command to be created, if an object command for the XtAppContext does not yet exist.

It returns:

appContext *(Command Name)* The name of the XtAppContext object command.

The equivalent C code for this subcommand would be:

```
appContext = XtWidgetToApplicationContext(widget);
```

The `augmentTranslations` Widget Subcommand

```
<widget> augmentTranslations <translations>
```

The `augmentTranslations` Widget subcommand provides an interface to the `XtAugmentTranslations` function. It takes:

translations *(List)* A translation table.

The equivalent C code for this subcommand would be:

```
XtAugmentTranslations(widget, translations);
```

The `disownSelection` Widget Subcommand

```
<widget> disownSelection <selection> <time>
```

The `disownSelection` Widget subcommand provides an interface to the `XtDisownSelection` function. It takes:

selection *(Atom)* A selection.

time *(Integer)* A time.

The equivalent C code for this subcommand would be:

```
XtDisownSelection(widget, selection, time);
```

The `getApplicationResources` Widget Subcommand

```
<widget> getApplicationResources [<resource>]...
  ==> <resources>
```

The `getApplicationResources` Widget subcommand provides an interface to the `XtGetApplicationResources` function. It takes:

resource *(String)* A resource name. If it starts with a hyphen, the hypen is removed.

It returns:

resources *(List)* The application resources.

The equivalent C code for this subcommand would be:

```
XtGetApplicationResources(widget, base, res, numRes,
                          args, numArgs);
```

The `getSelectionRequest` Widget Subcommand

```
<widget> getSelectionRequest <selection> [<eventName>] \
    [<requestId>]
  ==> <event>
```

The `getSelectionRequest` Widget subcommand provides an interface to the `XtGetSelectionRequest` function. It takes:

selection *(Atom)* A selection.

eventName *(String)* An event name.

requestId *(String)* A request id.

It returns:

event *(Command Name)* The name of the event object command.

The equivalent C code for this subcommand would be:

```
id = XtGetSelectionRequest(widget, selection, requestId);
```

The `getSelectionValue` Widget Subcommand

```
<widget> getSelectionValue <selection> <target> \
    <subcommand> <time>
```

The `getSelectionValue` Widget subcommand provides an interface to the `XtGetSelectionValue` function. It takes:

selection *(Atom)* A selection.

target *(Atom)* A target.

command *(Command String)* A command string to be evaluated with the value. The command is evaluated in a procedure body with the following local variables bound:

> **widget** *(Command Name)* The widget.
> **selection** *(Atom)* The selection.
> **type** *(Atom)* The type.
> **value** *(String)* The value.
> **format** *(String)* The format.

time *(Integer)* A time.

The equivalent C code for this subcommand would be:

```
XtGetSelectionValue(widget, selection, target,
                    callback, closure, time);
```

The `getSelectionValueIncremental` Widget Subcommand

```
<widget> getSelectionValueIncremental <selection> <target> \
    <subcommand> <time>
```

The `getSelectionValueIncremental` Widget subcommand provides an interface to the `XtGetSelectionValueIncremental` function. It takes:

selection *(Atom)* A selection.

target *(Atom)* A target.

command *(Command String)* A command string to be evaluated with the value. The command is evaluated in a procedure body with the following local variables bound:

> **widget** *(Command Name)* The widget.
>
> **selection** *(Atom)* The selection.
>
> **type** *(Atom)* The type.
>
> **value** *(String)* The value.

time *(Integer)* A time.

The equivalent C code for this subcommand would be:

```
XtGetSelectionValueIncremental(widget, selection, target,
                               selectionCallback, closure, time);
```

The `getSelectionValues` Widget Subcommand

```
<widget> getSelectionValues <selection> <targets> \
    <subcommands> <time>
```

The `getSelectionValues` Widget subcommand provides an interface to the `XtGetSelectionValues` function. It takes:

selection *(Atom)* A selection.

targets *(List)* A list of targets.

commands *(List)* A list of command strings to be evaluated with the values. The commands are evaluated in a procedure body with the following local variables bound:

> **widget** *(Command Name)* The widget.
>
> **selection** *(Atom)* The selection.
>
> **type** *(Atom)* The type.
>
> **value** *(String)* The value.

time *(Integer)* A time.

The equivalent C code for this subcommand would be:

```
XtGetSelectionValues(widget, selection, targets, count,
                     callback, closures, time);
```

The `getSelectionValuesIncremental` Widget Subcommand

```
<widget> getSelectionValuesIncremental <selection> <targets> \
    <subcommands> <time>
```

The `getSelectionValuesIncremental` Widget subcommand provides an interface to the `XtGetSelectionValuesIncremental` function. It takes:

selection *(Atom)* A selection.

targets *(List)* A list of targets.

commands *(List)* A list of command strings to be evaluated with the values. The commands are evaluated in a procedure body with the following local variables bound:

> **widget** *(Command Name)* The widget.
>
> **selection** *(Atom)* The selection.
>
> **type** *(Atom)* The type.
>
> **value** *(String)* The value.

time *(Integer)* A time.

The equivalent C code for this subcommand would be:

```
XtGetSelectionValuesIncremental(widget, selection, targets, count,
                                callback, closures, time);
```

The `installAccelerators` Widget Subcommand

```
<widget> installAccelerators <source>
```

The `installAccelerators` Widget subcommand provides an interface to the `XtInstallAccelerators` function. It takes:

source *(Command Name)* A widget object command name.

The equivalent C code for this subcommand would be:

```
XtInstallAccelerators(widget, source);
```

The `installAllAccelerators` Widget Subcommand

```
<widget> installAllAccelerators <source>
```

The `installAllAccelerators` Widget subcommand provides an interface to the `XtInstallAllAccelerators` function. It takes:

source *(Command Name)* A widget object command name.

The equivalent C code for this subcommand would be:

```
XtInstallAllAccelerators(widget, source);
```

The `isRealized` Widget Subcommand

```
<widget> isRealized
  ==> <isRealized>
```

The `isRealized` Widget subcommand provides an interface to the `XtIsRealized` function. It returns:

isRealized *(Boolean)* 1 if the widget is realized, 0 otherwise.

The equivalent C code for this subcommand would be:

```
isRealized = XtIsRealized(widget);
```

The `isSensitive` Widget Subcommand

```
<widget> isSensitive
  ==> <isSensitive>
```

The `isSensitive` Widget subcommand provides an interface to the `XtIsSensitive` function. It returns:

isSensitive *(Boolean)* 1 if the widget is sensitive, 0 otherwise.

The equivalent C code for this subcommand would be:

```
isSensitive = XtIsSensitive(widget);
```

The `isSubclass` Widget Subcommand

```
<widget> isSubclass <widgetClass>
  ==> <isSubclass>
```

The `isSubclass` Widget subcommand provides an interface to the `XtIsSubclass` function. It takes:

widgetClass *(Command Name)* A WidgetClass object command name.

It returns:

isSubclass *(Boolean)* 1 if the widget class of the widget is a subclass of the widget class provided, 0 otherwise.

The equivalent C code for this subcommand would be:

```
isSubclass = XtIsSubclass(widget, widgetClass);
```

The `map` Widget Subcommand

```
<widget> map
```

The `map` Widget subcommand provides an interface to the `XtMapWidget` function. The equivalent C code for this subcommand would be:

```
XtMapWidget(widget);
```

The `overrideTranslations` Widget Subcommand

```
<widget> overrideTranslations <translations>
```

The `overrideTranslations` Widget subcommand provides an interface to the `XtOverrideTranslations` function. It takes:

translations *(List)* A translation table.

The equivalent C code for this subcommand would be:

```
XtOverrideTranslations(widget, translations);
```

The `ownSelection` Widget Subcommand

```
<widget> ownSelection <selection> <time> \
    <convertSubcommand> <loseSubcommand>
  ==> <owner>
```

The `ownSelection` Widget subcommand provides an interface to the `XtOwnSelection` function. It takes:

selection *(Atom)* A selection.

time *(Integer)* A time.

convertCommand *(Command String)* A command string to be evaluated to perform the conversion. The command is evaluated in a procedure body with the following local variables bound:

> **widget** *(Command Name)* The widget.
>
> **selection** *(Atom)* The selection.
>
> **target** *(Atom)* The target.

loseCommand *(Command String)* A command string to be evaluated when the selection is lost. The command is evaluated in a procedure body with the following local variables bound:

> **widget** *(Command Name)* The widget.
>
> **selection** *(Atom)* The selection.

It returns:

owner *(Boolean)*

The equivalent C code for this subcommand would be:

```
owner = XtOwnSelection(widget, selection, time, convert,
                       lose, done);
```

The `ownSelectionIncremental` Widget Subcommand

```
<widget> ownSelectionIncremental <selection> <time> \
    <convertSubcommand> <cancelSubcommand> <loseSubcommand>
  ==> <owner>
```

The `ownSelectionIncremental` Widget subcommand provides an interface to the `XtOwnSelectionIncremental` function. It takes:

selection *(Atom)* A selection.

time *(Integer)* A time.

convertCommand *(Command String)* A command string to be evaluated to perform the conversion. The command is evaluated in a procedure body with the following local variables bound:

- **widget** *(Command Name)* The widget.
- **selection** *(Atom)* The selection.
- **target** *(Atom)* The target.
- **max_length** *(Integer)* The maximum length.
- **receiver_id** *(Atom)* The receiver id.

cancelCommand *(Command String)* A command string to be evaluated when the conversion is cancelled. The command is evaluated in a procedure body with the following local variables bound:

- **widget** *(Command Name)* The widget.
- **selection** *(Atom)* The selection.
- **target** *(Atom)* The target.
- **max_length** *(Integer)* The receiver id.

loseCommand *(Command String)* A command string to be evaluated when the selection is lost. The command is evaluated in a procedure body with the following local variables bound:

- **widget** *(Command Name)* The widget.
- **selection** *(Atom)* The selection.

It returns:

owner *(Boolean)*

The equivalent C code for this subcommand would be:

```
owner = XtOwnSelectionIncremental(widget, selection, time,
                                  convertCallback, loseCallback,
                                  doneCallback, cancelCallback,
                                  (XtPointer)sic);
```

The `parent` Widget Subcommand

```
<widget> parent
  ==> <parent>
```

The `parent` Widget subcommand provides an interface to the `XtParent` function. It returns:

parent *(Command Name)* The name of the parent widget object command.

The equivalent C code for this subcommand would be:

```
parent = XtParent(widget);
```

The `realize` Widget Subcommand

```
<widget> realize
```

The `realize` Widget subcommand provides an interface to the `XtRealizeWidget` function. The equivalent C code for this subcommand would be:

```
XtRealizeWidget(widget);
```

The `setKeyboardFocus` Widget Subcommand

```
<widget> setKeyboardFocus <descendent>
```

The `setKeyboardFocus` Widget subcommand provides an interface to the `XtSetKeyboardFocus` function. It takes:

descendent *(Command Name)* The name of the descendent Widget object command.

The equivalent C code for this subcommand would be:

```
XtSetKeyboardFocus(widget, descendent);
```

The `setMappedWhenManaged` Widget Subcommand

```
<widget> setMappedWhenManaged <mappedWhenManaged>
```

The `setMappedWhenManaged` Widget subcommand provides an interface to the `XtSetMappedWhenManaged` function. It takes:

mappedWhenManaged *(Boolean)*

The equivalent C code for this subcommand would be:

```
XtSetMappedWhenManaged(widget, mappedWhenManaged);
```

The `setSensitive` Widget Subcommand

```
<widget> setSensitive <sensitive>
```

The `setSensitive` Widget subcommand provides an interface to the `XtSetSensitive` function. It takes:

sensitive *(Boolean)*

The equivalent C code for this subcommand would be:

```
XtSetSensitive(widget, sensitive);
```

The `translateCoords` Widget Subcommand

```
<widget> translateCoords <pt>
  ==> <pt>
```

The `translateCoords` Widget subcommand provides an interface to the `XtTranslateCoords` function. It returns:

pt *(List)*

The equivalent C code for this subcommand would be:

```
XtTranslateCoords(widget, x, y, &rootx, &rooty);
```

The `uninstallTranslations` Widget Subcommand

```
<widget> uninstallTranslations
```

The `uninstallTranslations` Widget subcommand provides an interface to the `XtUninstallTranslations` function. The equivalent C code for this subcommand would be:

```
XtUninstallTranslations(widget);
```

The `unMap` Widget Subcommand

```
<widget> unMap
```

The `unMap` Widget subcommand provides an interface to the `XtUnmapWidget` function. The equivalent C code for this subcommand would be:

```
XtUnmapWidget(widget);
```

The `unrealize` Widget Subcommand

```
<widget> unrealize
```

The `unrealize` Widget subcommand provides an interface to the `XtUnrealizeWidget` function. The equivalent C code for this subcommand would be:

```
XtUnrealizeWidget(widget);
```

The `addCallback` Widget Subcommand

```
<widget> addCallback <name> <subcommand>
```

The `addCallback` Widget subcommand provides an interface to the `XtAddCallback` function. It takes:

name *(String)* A callback name.

command *(Command String)* A command string to be evaluated with the value. The command is evaluated in a procedure body with the following local variables bound:

> **widget** *(Command Name)* The widget.

The equivalent C code for this subcommand would be:

```
XtAddCallback(widget, name, proc, clientData);
```

The `callCallbacks` Widget Subcommand

```
<widget> callCallbacks <name>
```

The `callCallbacks` Widget subcommand provides an interface to the `XtCallCallbacks` function. It takes:

name *(String)* A callback name.

The equivalent C code for this subcommand would be:

```
XtCallCallbacks(widget, name);
```

The `hasCallbacks` Widget Subcommand

```
<widget> hasCallbacks <name>
  ==> <n>
```

The `hasCallbacks` Widget subcommand provides an interface to the `XtHasCallbacks` function. It takes:

name *(String)* A callback name.

It returns:

n *(Integer)*

The equivalent C code for this subcommand would be:

```
n = XtHasCallbacks(widget, name);
```

The `removeCallback` Widget Subcommand

```
<widget> removeCallback <name> <subcommand>
```

The `removeCallback` Widget subcommand provides an interface to the `XtRemoveCallback` function. It takes:

name *(String)* A callback name.

command *(Command String)* A command string for matching.

The equivalent C code for this subcommand would be:

```
XtRemoveCallback(widget, name, proc, clientData);
```

The `removeAllCallbacks` Widget Subcommand

```
<widget> removeAllCallbacks <name>
```

The `removeAllCallbacks` Widget subcommand provides an interface to the `XtRemoveAllCallbacks` function. It takes:

name *(String)* A callback name.

The equivalent C code for this subcommand would be:

```
XtRemoveAllCallbacks(widget, name);
```

D.13 Composite Subcommands

The `manageChild` Composite Subcommand

```
<composite> manageChild <child>
```

The `manageChild` Composite subcommand provides an interface to the `XtManageChild` function. It takes:

child *(Command Name)* A name of a child Widget object command.

The equivalent C code for this subcommand would be:

```
XtManageChild(child);
```

The `manageChildren` Composite Subcommand

```
<composite> manageChildren <children>
```

The `manageChildren` Composite subcommand provides an interface to the `XtManageChildren` function. It takes:

children *(List)* A list of child Widget object commands.

The equivalent C code for this subcommand would be:

```
XtManageChildren(children, numChildren);
```

The `unmanageChild` Composite Subcommand

```
<composite> unmanageChild <child>
```

The `unmanageChild` Composite subcommand provides an interface to the `XtUnmanageChild` function. It takes:

child *(Command Name)* A name of a child Widget object command.

The equivalent C code for this subcommand would be:

```
XtUnmanageChild(child);
```

The `unmanageChildren` Composite Subcommand

```
<composite> unmanageChildren <children>
```

The `unmanageChildren` Composite subcommand provides an interface to the `XtUnmanageChildren` function. It takes:

children *(List)* A list of child Widget object commands.

The equivalent C code for this subcommand would be:

```
XtUnmanageChildren(children, numChildren);
```

D.14 Shell Subcommands

The popdown Shell Subcommand

```
<shell> popdown
```

The `popdown` Shell subcommand provides an interface to the `XtPopdown` function. The equivalent C code for this subcommand would be:

```
XtPopdown(popupShell);
```

The popup Shell Subcommand

```
<shell> popup <grabKind>
```

The `popup` Shell subcommand provides an interface to the `XtPopup` function. It takes:

grabKind *(String)* One of `none`, `nonexclusive`, or `exclusive`.

The equivalent C code for this subcommand would be:

```
XtPopup(popupShell, grabKind);
```

The popupSpringLoaded Shell Subcommand

```
<shell> popupSpringLoaded
```

The `popupSpringLoaded` Shell subcommand provides an interface to the `XtPopupSpringLoaded` function. The equivalent C code for this subcommand would be:

```
XtPopupSpringLoaded(popupShell);
```

D.15 AsciiSource Subcommands

The freeString AsciiSource Subcommand

```
<asciiSource> freeString
```

The `freeString` AsciiSource subcommand provides an interface to the `XawAsciiSourceFreeString` function. The equivalent C code for this subcommand would be:

```
XawAsciiSourceFreeString(w);
```

The save AsciiSource Subcommand

```
<asciiSource> save
  ==> <successful>
```

The `save` AsciiSource subcommand provides an interface to the `XawAsciiSave` function. It returns:

successful *(Boolean)* 1 if the file was saved sucessfully, 0 otherwise.

The equivalent C code for this subcommand would be:

```
result = XawAsciiSave(w);
```

The `saveAsFile` AsciiSource Subcommand

```
<asciiSource> saveAsFile <name>
  ==> <successful>
```

The `saveAsFile` AsciiSource subcommand provides an interface to the `XawAsciiSaveAsFile` function. It takes:

name *(String)* A file name.

It returns:

successful *(Boolean)* 1 if the file was saved sucessfully, 0 otherwise.

The equivalent C code for this subcommand would be:

```
successful = XawAsciiSaveAsFile(w, name);
```

The `changed` AsciiSource Subcommand

```
<asciiSource> changed
  ==> <changed>
```

The `changed` AsciiSource subcommand provides an interface to the `XawAsciiSourceChanged` function. It returns:

changed *(Boolean)* 1 if the buffer has been modified, 0 otherwise.

The equivalent C code for this subcommand would be:

```
changed = XawAsciiSourceChanged(w);
```

D.16 Dialog Subcommands

The `addButton` Dialog Subcommand

```
<dialog> addButton <name> <subcommand>
```

The `addButton` Dialog subcommand provides an interface to the `XawDialogAddButton` function. It takes:

name *(String)* A button name.

command *(Command String)* A command string to be evaluated when the button is pressed. The command is evaluated in a procedure body with the following local variables bound:

> **widget** *(Command Name)* The widget.
>
> **call_data** *(Command Name)* The call data.

The equivalent C code for this subcommand would be:

```
XawDialogAddButton(dialog, name, function, clientData);
```

The `getValueString` Dialog Subcommand

```
<dialog> getValueString
  ==> <value>
```

The `getValueString` Dialog subcommand provides an interface to the `XawDialogGetValueString` function. It returns:

value *(String)* The value.

The equivalent C code for this subcommand would be:

```
value = XawDialogGetValueString(w);
```

D.17 Form Subcommands

The `doLayout` Form Subcommand

```
<form> doLayout <doLayout>
```

The `doLayout` Form subcommand provides an interface to the `XawFormDoLayout` function. It takes:

doLayout *(String)* A boolean value.

The equivalent C code for this subcommand would be:

```
XawFormDoLayout(w, doLayout);
```

D.18 List Subcommands

The `change` List Subcommand

```
<list> change <list> <resize>
```

The `change` List subcommand provides an interface to the `XawListChange` function. It takes:

list *(List)* A new list of values.

resize *(Boolean)* 1 to resize, 0 otherwise.

The equivalent C code for this subcommand would be:

```
XawListChange(w, list, nitems, longest, resize);
```

The `unhighlight` List Subcommand

```
<list> unhighlight
```

The `unhighlight` List subcommand provides an interface to the `XawListUnhighlight` function. The equivalent C code for this subcommand would be:

```
XawListUnhighlight(w);
```

The `highlight` List Subcommand

```
<list> highlight <item>
```

The **highlight** List subcommand provides an interface to the **XawListHighlight** function. It takes:

item *(Integer)* The item number in the list.

The equivalent C code for this subcommand would be:

```
XawListHighlight(w, item);
```

The `showCurrent` List Subcommand

```
<list> showCurrent
  ==> <element>
```

The **showCurrent** List subcommand provides an interface to the **XawListShowCurrent** function. It returns:

element *(String)* The current element in the list.

The equivalent C code for this subcommand would be:

```
element = XawListShowCurrent(w);
```

D.19 Paned Subcommands

The `setMinMax` Paned Subcommand

```
<paned> setMinMax <min> <max>
```

The **setMinMax** Paned subcommand provides an interface to the **XawPanedSetMinMax** function. It takes:

min *(Integer)* A minimum.

max *(Integer)* A maximum.

The equivalent C code for this subcommand would be:

```
XawPanedSetMinMax(w, min, max);
```

The `getMinMax` Paned Subcommand

```
<paned> getMinMax
  ==> <min> <max>
```

The **getMinMax** Paned subcommand provides an interface to the **XawPanedGetMinMax** function. It returns:

min *(Integer)* The minimum.

max *(Integer)* The maximum.

The equivalent C code for this subcommand would be:

```
XawPanedGetMinMax(w, &minReturn, &maxReturn);
```

The `setRefigureMode` Paned Subcommand

```
<paned> setRefigureMode <mode>
```

The `setRefigureMode` Paned subcommand provides an interface to the `XawPanedSetRefigureMode` function. It takes:

> **mode** *(Boolean)* A refigure mode.

The equivalent C code for this subcommand would be:

```
XawPanedSetRefigureMode(w, mode);
```

The `getNumSub` Paned Subcommand

```
<paned> getNumSub
  ==> <sumSub>
```

The `getNumSub` Paned subcommand provides an interface to the `XawPanedGetNumSub` function. It returns:

> **numSub** *(Integer)* The numSub.

The equivalent C code for this subcommand would be:

```
numSub = XawPanedGetNumSub(w);
```

The `allowResize` Paned Subcommand

```
<paned> allowResize
```

The `allowResize` Paned subcommand provides an interface to the `XawPanedAllowResize` function. The equivalent C code for this subcommand would be:

```
XawPanedAllowResize(w, allowResize);
```

D.20 Scrollbar Subcommands

The `setThumb` Scrollbar Subcommand

```
<scrollbar> setThumb <top> <shown>
```

The `setThumb` Scrollbar subcommand provides an interface to the `XawScrollbarSetThumb` function. It takes:

> **top** *(Float)* A value between 0 and 1.
>
> **shown** *(Float)* A value between 0 and 1.

The equivalent C code for this subcommand would be:

```
XawScrollbarSetThumb(scrollbar, top, shown);
```

D.21 SimpleMenu Subcommands

The `getActiveEntry` SimpleMenu Subcommand

```
<simpleMenu> getActiveEntry
  ==> <widget>
```

The `getActiveEntry` SimpleMenu subcommand provides an interface to the `XawSimpleMenuGetActiveEntry` function. It returns:

widget *(Command Name)* The name of the active entry Widget object command.

The equivalent C code for this subcommand would be:

```
widget = XawSimpleMenuGetActiveEntry(w);
```

The `clearActiveEntry` SimpleMenu Subcommand

```
<simpleMenu> clearActiveEntry
```

The `clearActiveEntry` SimpleMenu subcommand provides an interface to the `XawSimpleMenuClearActiveEntry` function. The equivalent C code for this subcommand would be:

```
XawSimpleMenuClearActiveEntry(w);
```

D.22 Text Subcommands

The `display` Text Subcommand

```
<text> display
```

The `display` Text subcommand provides an interface to the `XawTextDisplay` function. The equivalent C code for this subcommand would be:

```
XawTextDisplay(w);
```

The `enableRedisplay` Text Subcommand

```
<text> enableRedisplay
```

The `enableRedisplay` Text subcommand provides an interface to the `XawTextEnableRedisplay` function. The equivalent C code for this subcommand would be:

```
XawTextEnableRedisplay(w);
```

The `disableRedisplay` Text Subcommand

```
<text> disableRedisplay
```

The `disableRedisplay` Text subcommand provides an interface to the `XawTextDisableRedisplay` function. The equivalent C code for this subcommand would be:

```
XawTextDisableRedisplay(w);
```

The `setSelectionArray` Text Subcommand

```
<text> setSelectionArray <sarray>
```

The `setSelectionArray` Text subcommand provides an interface to the `XawTextSetSelectionArray` function. It takes:

sarray *(List)* A list of selection types. A selection type is one of `null`, `position`, `char`, `word`, `line`, `paragraph`, or `all`.

The equivalent C code for this subcommand would be:

```
XawTextSetSelectionArray(w, sarray);
```

The `getSelectionPos` Text Subcommand

```
<text> getSelectionPos
  ==> <begin> <end>
```

The `getSelectionPos` Text subcommand provides an interface to the `XawTextGetSelectionPos` function. It returns:

begin *(Integer)* The beginning of the selection.

end *(Integer)* The end of the selection.

The equivalent C code for this subcommand would be:

```
XawTextGetSelectionPos(w, &begin, &end);
```

The `setSource` Text Subcommand

```
<text> setSource <source> <position>
```

The `setSource` Text subcommand provides an interface to the `XawTextSetSource` function. It takes:

source *(Command Name)* A name of a TextSource object command.

position *(Integer)*

The equivalent C code for this subcommand would be:

```
XawTextSetSource(w, source, position);
```

The `replace` Text Subcommand

```
<text> replace <start> <end> <text>
  ==> <status>
```

The `replace` Text subcommand provides an interface to the `XawTextReplace` function. It takes:

start *(Integer)* The start of the region in the buffer to be replaced.

end *(Integer)* The end of the region in the buffer to be replaced.

text *(String)* The text to replace the region in the buffer.

It returns:

status *(Integer)* The status.

The equivalent C code for this subcommand would be:

```
status = XawTextReplace(w, start, end, &text);
```

The `topPosition` Text Subcommand

```
<text> topPosition
  ==> <position>
```

The `topPosition` Text subcommand provides an interface to the `XawTextTopPosition` function. It returns:

> **position** *(Integer)* The position of the top of the window in the buffer.

The equivalent C code for this subcommand would be:

```
position = XawTextTopPosition(w);
```

The `setInsertionPoint` Text Subcommand

```
<text> setInsertionPoint <position>
```

The `setInsertionPoint` Text subcommand provides an interface to the `XawTextSetInsertionPoint` function. It takes:

> **position** *(Integer)* A position in the buffer.

The equivalent C code for this subcommand would be:

```
XawTextSetInsertionPoint(w, position);
```

The `getInsertionPoint` Text Subcommand

```
<text> getInsertionPoint
  ==> <position>
```

The `getInsertionPoint` Text subcommand provides an interface to the `XawTextGetInsertionPoint` function. It returns:

> **position** *(Integer)* The position of the insertion point in the buffer.

The equivalent C code for this subcommand would be:

```
position = XawTextGetInsertionPoint(w);
```

The `unsetSelection` Text Subcommand

```
<text> unsetSelection
```

The `unsetSelection` Text subcommand provides an interface to the `XawTextUnsetSelection` function. The equivalent C code for this subcommand would be:

```
XawTextUnsetSelection(w);
```

The `setSelection` Text Subcommand

```
<text> setSelection <left> <right>
```

The `setSelection` Text subcommand provides an interface to the `XawTextSetSelection` function. It takes:

left *(Integer)* A starting text position.

right *(Integer)* An ending text position.

The equivalent C code for this subcommand would be:

```
XawTextSetSelection(w, left, right);
```

The `invalidate` Text Subcommand

```
<text> invalidate <from> <to>
```

The `invalidate` Text subcommand provides an interface to the `XawTextInvalidate` function. It takes:

from *(Integer)* A starting text position.

to *(Integer)* An ending text position.

The equivalent C code for this subcommand would be:

```
XawTextInvalidate(w, from, to);
```

The `getSource` Text Subcommand

```
<text> getSource
  ==> <widget>
```

The `getSource` Text subcommand provides an interface to the `XawTextGetSource` function. It returns:

widget *(Command Name)* The name of the TextSource object command.

The equivalent C code for this subcommand would be:

```
widget = XawTextGetSource(w);
```

The `search` Text Subcommand

```
<text> search <dir> <text>
  ==> <position>
```

The `search` Text subcommand provides an interface to the `XawTextSearch` function. It takes:

dir *(String)* One of `left` or `right`.

text *(String)* The search key.

It returns:

position *(Integer)*

The equivalent C code for this subcommand would be:

```
position = XawTextSearch(w, dir, &text);
```

The `displayCaret` Text Subcommand

```
<text> displayCaret <visible>
```

The `displayCaret` Text subcommand provides an interface to the `XawTextDisplayCaret` function. It takes:

visible *(Boolean)* 1 to make the caret visible, 0 otherwise.

The equivalent C code for this subcommand would be:

```
XawTextDisplayCaret(w, visible);
```

D.23 TextSink Subcommands

The `displayText` TextSink Subcommand

```
<textSink> displayText <position> <textPosition1> <textPosition2> \
    <highlight>
```

The `displayText` TextSink subcommand provides an interface to the `XawTextSinkDisplayText` function. It takes:

position *(List)* A list of two cursor coordinates.

textPosition1 *(Integer)* A text position.

textPosition2 *(Integer)* A text position.

highlight *(Boolean)* 1 to highlight, 0 otherwise.

The equivalent C code for this subcommand would be:

```
XawTextSinkDisplayText(w, x, y, pos1, pos2, highlight);
```

The `insertCursor` TextSink Subcommand

```
<textSink> insertCursor <position> <state>
```

The `insertCursor` TextSink subcommand provides an interface to the `XawTextSinkInsertCursor` function. It takes:

position *(Integer)* A list of two cursor coordinates.

state *(String)* One of `on` or `off`.

The equivalent C code for this subcommand would be:

```
XawTextSinkInsertCursor(w, x, y, state);
```

The `clearToBackground` TextSink Subcommand

```
<textSink> clearToBackground <rect>
```

The `clearToBackground` TextSink subcommand provides an interface to the `XawTextSinkClearToBackground` function. It takes:

rect *(List)* A list of four numbers: x, y, w, h.

The equivalent C code for this subcommand would be:

```
XawTextSinkClearToBackground(w, rect.x, rect.y,
                             rect.width, rect.height);
```

The `findPosition` TextSink Subcommand

```
<textSink> findPosition <fromPos> <fromX> <width> <stopAtWordBreak>
  ==> <position> <width> <height>
```

The `findPosition` TextSink subcommand provides an interface to the `XawTextSinkFindPosition` function. It takes:

fromPos *(Integer)* A text position.

fromX *(Integer)* A from x coordinate.

width *(Integer)* A width.

stopAtWordBreak *(Boolean)*

It returns:

position *(Integer)* The position.

width *(Integer)* The width.

height *(Integer)* The height.

The equivalent C code for this subcommand would be:

```
XawTextSinkFindPosition(w, fromPos, fromX, width, stopAtWordBreak,
                        &pos, &widthReturn, &height);
```

The `findDistance` TextSink Subcommand

```
<textSink> findDistance <fromPos> <fromX> <toPos>
  ==> <height>
```

The `findDistance` TextSink subcommand provides an interface to the `XawTextSinkFindDistance` function. It takes:

fromPos *(Integer)* A text position.

fromX *(Integer)* A from x coordinate.

toPos *(Integer)* A text position.

It returns:

height *(Integer)* The height.

The equivalent C code for this subcommand would be:

```
XawTextSinkFindDistance(w, fromPos, fromX, toPos,
                        &width, &pos, &height);
```

The `resolve` TextSink Subcommand

```
<textSink> resolve <fromPos> <fromX> <width>
  ==> <position>
```

The `resolve` TextSink subcommand provides an interface to the `XawTextSinkResolve` function. It takes:

fromPos *(Integer)* A text position.

fromX *(Integer)* A from x coordinate.

width *(Integer)* A width.

It returns:

position *(Integer)* The text position.

The equivalent C code for this subcommand would be:

```
XawTextSinkResolve(w, fromPos, fromX, width, &pos);
```

The maxLines TextSink Subcommand

```
<textSink> maxLines <height>
  ==> <lines>
```

The `maxLines` TextSink subcommand provides an interface to the `XawTextSinkMaxLines` function. It takes:

height *(Integer)* A height.

It returns:

lines *(Integer)* The number of lines.

The equivalent C code for this subcommand would be:

```
lines = XawTextSinkMaxLines(w, height);
```

The maxHeight TextSink Subcommand

```
<textSink> maxHeight <lines>
  ==> <height>
```

The `maxHeight` TextSink subcommand provides an interface to the `XawTextSinkMaxHeight` function. It takes:

lines *(Integer)* A number of lines.

It returns:

height *(Integer)* The height.

The equivalent C code for this subcommand would be:

```
height = XawTextSinkMaxHeight(w, lines);
```

The setTabs TextSink Subcommand

```
<textSink> setTabs <tabs>
```

The `setTabs` TextSink subcommand provides an interface to the `XawTextSinkSetTabs` function. It takes:

tabs *(List)* A list of integer tab stops.

The equivalent C code for this subcommand would be:

```
XawTextSinkSetTabs(w, tabCount, tabs);
```

The getCursorBounds TextSink Subcommand

```
<textSink> getCursorBounds
  ==> <rect>
```

The `getCursorBounds` TextSink subcommand provides an interface to the `XawTextSinkGetCursorBounds` function. It returns:

rect *(List)* The cursor bounds as a list of four numbers: x, y, w, h.

The equivalent C code for this subcommand would be:

```
XawTextSinkGetCursorBounds(w, &rect);
```

D.24 TextSource Subcommands

The `read` TextSource Subcommand

```
<textSource> read <positionVar> <length>
  ==> <text>
```

The `read` TextSource subcommand provides an interface to the `XawTextSourceRead` function. It takes:

positionVar *(Variable Name)* A variable holding a position value that is updated as the read progresses.

length *(Integer)* The number if bytes to read.

It returns:

text *(String)* The text read.

The equivalent C code for this subcommand would be:

```
pos = XawTextSourceRead(w, pos, &text, length);
```

The `replace` TextSource Subcommand

```
<textSource> replace <start> <end> <text>
  ==> <status>
```

The `replace` TextSource subcommand provides an interface to the `XawTextSourceReplace` function. It takes:

start *(Integer)* A starting text position.

end *(Integer)* An ending text position.

text *(String)* The text to replace the region in the buffer.

It returns:

status *(Integer)* The status.

The equivalent C code for this subcommand would be:

```
status = XawTextSourceReplace(w, start, end, &text);
```

The `scan` TextSource Subcommand

```
<textSource> scan <position> <type> <dir> <count> <include>
  ==> <position>
```

The `scan` TextSource subcommand provides an interface to the `XawTextSourceScan` function. It takes:

position *(Integer)* A text position.

type *(String)* One of `positions`, `whiteSpace`, `eol`, `paragraph`, or `all`.

dir *(String)* One of `left` or `right`.

count *(Integer)* A count.

include *(Boolean)*

It returns:

position *(Integer)* The position.

The equivalent C code for this subcommand would be:

```
position = XawTextSourceScan(w, position, type, dir,
                             count, include);
```

The `search` TextSource Subcommand

```
<textSource> search <position> <dir> <text>
  ==> <position>
```

The `search` TextSource subcommand provides an interface to the `XawTextSourceSearch` function. It takes:

dir *(String)* One of `left` or `right`.

text *(String)* The search key.

It returns:

position *(Integer)* The position.

The equivalent C code for this subcommand would be:

```
position = XawTextSourceSearch(w, position, dir, &text);
```

The `convertSelection` TextSource Subcommand

```
<textSource> convertSelection <selection> <target> <type>
  ==> <converted>
```

The `convertSelection` TextSource subcommand provides an interface to the `XawTextSourceConvertSelection` function. It takes:

selection *(Atom)* A selection.

target *(Atom)* A target.

type *(Type)* A type.

It returns:

converted *(Boolean)* 1 if successful, 0 otherwise.

The equivalent C code for this subcommand would be:

```
converted = XawTextSourceConvertSelection(w, &selection, &target,
                                          &type, &value, &length,
                                          &format);
```

The `setSelection` TextSource Subcommand

```
<textSource> setSelection <start> <end> <selection>
```

The `setSelection` TextSource subcommand provides an interface to the `XawTextSourceSetSelection` function. It takes:

start *(Integer)* A starting text position.

end *(Integer)* An ending text position.

selection *(Atom)* A selection.

The equivalent C code for this subcommand would be:

```
XawTextSourceSetSelection(w, start, end, selection);
```

D.25 Toggle Subcommands

The changeRadioGroup Toggle Subcommand

```
<toggle> changeRadioGroup <radioGroup>
```

The `changeRadioGroup` Toggle subcommand provides an interface to the `XawToggleChangeRadioGroup` function. It takes:

radioGroup *(Command Name)* A name of a group Widget object command.

The equivalent C code for this subcommand would be:

```
XawToggleChangeRadioGroup(w, radioGroup);
```

The getCurrent Toggle Subcommand

```
<toggle> getCurrent
  ==> <group>
```

The `getCurrent` Toggle subcommand provides an interface to the `XawToggleGetCurrent` function. It returns:

group *(Command Name)* The name of the current Widget object command.

The equivalent C code for this subcommand would be:

```
group = XawToggleGetCurrent(radioGroup);
```

The setCurrent Toggle Subcommand

```
<toggle> setCurrent <radioData>
```

The `setCurrent` Toggle subcommand provides an interface to the `XawToggleSetCurrent` function. It takes:

group *(Command Name)* A Widget object command name.

The equivalent C code for this subcommand would be:

```
XawToggleSetCurrent(radioGroup, radioData);
```

The unsetCurrent Toggle Subcommand

```
<toggle> unsetCurrent
```

The `unsetCurrent` Toggle subcommand provides an interface to the `XawToggleUnsetCurrent` function. The equivalent C code for this subcommand would be:

```
XawToggleUnsetCurrent(radioGroup);
```

D.26 Tree Subcommands

The `forceLayout` Tree Subcommand

```
<tree> forceLayout
```

The `forceLayout` Tree subcommand provides an interface to the `XawTreeForceLayout` function. The equivalent C code for this subcommand would be:

```
XawTreeForceLayout(tree);
```

D.27 Viewport Subcommands

The `setLocation` ViewPort Subcommand

```
<viewport> setLocation <location>
```

The `setLocation` ViewPort subcommand provides an interface to the `XawViewportSetLocation` function. It takes:

location *(List)* A list of two floats.

The equivalent C code for this subcommand would be:

```
XawViewportSetLocation(gw, xoff, yoff);
```

The `setCoordinates` ViewPort Subcommand

```
<viewport> setCoordinates <position>
```

The `setCoordinates` ViewPort subcommand provides an interface to the `XawViewportSetCoordinates` function. It takes:

position *(List)* A pair of coordinates: x, y.

The equivalent C code for this subcommand would be:

```
XawViewportSetCoordinates(gw, x, y);
```

Appendix E

The vspuzzle Example Application

E.1 The vsPuzzle.h In-Band Module Header File

```
#ifndef _VSPUZZLE_H_
#define _VSPUZZLE_H_

#ifdef __GNUG__
#pragma interface
#endif

extern "C" {
#include <stdlib.h>
}
#include <vs/vsEntity.h>
#include <vs/vsXdrBlock.h>
#include <vs/vsVideoFrame.h>
#include <vs/vsFilter.h>

extern "C" {
  extern int VsPuzzlePositionCmd(ClientData,Tcl_Interp*,int,char*[]);
  extern int VsPuzzleScrambleCmd(ClientData,Tcl_Interp*,int,char*[]);
}

class VsPuzzle :public VsFilter {
  int config[6][6], x, y, dim;
  Boolean solved;
  friend int VsPuzzlePositionCmd(ClientData,Tcl_Interp*,int,char*[]);
  friend int VsPuzzleScrambleCmd(ClientData,Tcl_Interp*,int,char*[]);
  static VsEntity* Creator(Tcl_Interp*,VsEntity*,const char*);
  static VsSymbol* classSymbol;
  VsPuzzle(const VsPuzzle&);
  VsPuzzle& operator=(const VsPuzzle&);
protected:
  virtual Boolean WorkRequiredP(VsPayload* p);
public:
  VsPuzzle(Tcl_Interp*, VsEntity*, const char*);
  virtual ~VsPuzzle();
  virtual VsSymbol* ClassSymbol() const { return classSymbol; };
  virtual void* ObjPtr(const VsSymbol*);
  virtual Boolean Work();
  static VsPuzzle* DerivePtr(VsObj*);
  static int Get(Tcl_Interp*, char*, VsPuzzle**);
  static void InitInterp(Tcl_Interp*);
};

inline VsPuzzle*
VsPuzzle::DerivePtr(VsObj* o) {
  return (VsPuzzle*)o->ObjPtr(classSymbol);
```

```
}

inline int
VsPuzzle::Get(Tcl_Interp* in, char* nm, VsPuzzle** pp) {
  return VsTclObj::Get(in, nm, classSymbol, (void**)pp);
}

#endif /* _VSPUZZLE_H_ */
```

E.2 The vsPuzzle.cc In-Band Module Source File

```
#ifdef __GNUG__
#pragma implementation
#endif

extern "C" {
#include <string.h>
}

#include <vs/vslib.h>
#include <vs/vsVideoFrame.h>
#include <vs/vsPuzzle.h>
#include <vs/vsTclClass.h>

int
VsPuzzlePositionCmd(ClientData cd,Tcl_Interp* in,int argc,char* argv[]){
  VsPuzzle* p = (VsPuzzle*)cd;

  /* Check parameter count */
  if (argc > 2 || argc < 1)
    return VsTclErrArgCnt(in, argv[0], "?position?");

  /* Set new position if supplied */
  if (argc == 2) {
    /* Get position parameter and check it */
    int x, y;
    if (VsGetIntPair(in, argv[1], &x, &y) != TCL_OK) return TCL_ERROR;
    if (x >= p->dim || x < 0)
      return VsTclErrBadVal(in, "x position within range", argv[1]);
    if (y >= p->dim || y < 0)
      return VsTclErrBadVal(in, "y position within range", argv[2]);
    if (x-p->x != 1 && x-p->x != -1 && y-p->y != 1 && y-p->y != -1)
      return VsTclErrBadVal(in, "x or y adjacent", "none");
    if (x-p->x != 0 && y-p->y != 0)
      return VsTclErrBadVal(in, "x or y adjacent", "both");

    /* Change the position */
    p->config[p->y][p->x] = p->config[y][x];
    p->x = x;
    p->y = y;
    p->config[p->y][p->x] = 0;

    /* Check if the puzzle is solved */
    p->solved = True;
    for (int k=0; k<p->dim*p->dim; k++)
      if (p->config[k/p->dim][k%p->dim]!=k) p->solved = False;
  }

  return VsReturnIntPair(in, p->x, p->y);
}

int
VsPuzzleScrambleCmd(ClientData cd,Tcl_Interp* in,int argc,char* argv[]){
  VsPuzzle* p = (VsPuzzle*)cd;

  /* Get the dimemsion parameter */
  int dim;
  if (argc != 2) return VsTclErrArgCnt(in, argv[0], "dimension");
  if (VsGetInt(in, argv[1], &dim) != TCL_OK) return TCL_ERROR;
```

```
  if (dim > 6 || dim < 3)
    return VsTclErrBadVal(in, "integer from 3 to 6", argv[1]);

  /* Change dimensions if necessary */
  if (p->dim != dim) {
    p->dim = dim;
    p->x = 0;
    p->y = 0;
  }

  /* Initialize The configuration matrix */
  for (int k = 0; k < dim*dim; k++)
    p->config[k/dim][k%dim] = k;

  /* Scramble the puzzle */
  for (k = 0; k < dim*dim; k++) {
    int x = rand() % dim;
    int y = rand() % dim;
    int swap = p->config[k/dim][k%dim];
    p->config[k/dim][k%dim] = p->config[x][y];
    p->config[x][y] = swap;
  }

  /* Figure out where the hole is */
  for (k = 0; k < dim*dim; k++)
    if (p->config[k/dim][k%dim]==0) {
      p->x = k%dim;
      p->y = k/dim;
    }

  /* Check if the puzzle is solved */
  p->solved = True;
  for (k=0; k<p->dim*p->dim; k++)
    if (p->config[k/p->dim][k%p->dim]!=k) p->solved = False;

  return VsReturnNull(in);
}

VsPuzzle::VsPuzzle(Tcl_Interp* in, VsEntity* pr, const char* nm)
  :VsFilter(in,pr,nm),dim(3),solved(True),x(0),y(0)
{
  CreateCommand("position",VsPuzzlePositionCmd,(ClientData)this,0);
  CreateCommand("scramble",VsPuzzleScrambleCmd,(ClientData)this,0);
  for (int i=0; i<dim*dim; i++)
      config[i/dim][i%dim]=i;
}

VsPuzzle::~VsPuzzle() {
}

void*
VsPuzzle::ObjPtr(const VsSymbol* cl) {
  return (cl == classSymbol)? this : VsFilter::ObjPtr(cl);
}

Boolean
VsPuzzle::WorkRequiredP(VsPayload *p) {
  return !solved && VsVideoFrame::DerivePtr(p) != 0;
}

Boolean
VsPuzzle::Work() {
  VsVideoFrame* frame = VsVideoFrame::DerivePtr(payload);

  if (!solved) {
    caddr_t image_data = frame->Data().Ptr();
    int dx = frame->Width()/dim;
    int dy = frame->Height()/dim;
    int bytesPerLine = frame->BytesPerLine();
    VsXdrBlock newData(frame->Data().Fore());
    caddr_t compute_data = newData.Ptr();
```

```
    for (int j=0; j<dim; j++) {
      for (int i=0; i<dim; i++) {
        int c = config[j][i];
        if (c == 0) {
          caddr_t dst = compute_data + j*dy*bytesPerLine + i*dx;
          caddr_t dstEnd = dst + dy*bytesPerLine;
          do {
            memset(dst, 16, dx);
            dst += bytesPerLine;
          } while (dst != dstEnd);
        } else {
          caddr_t src=image_data+c/dim*dy*bytesPerLine+c%dim*dx;
          caddr_t dst = compute_data + j*dy*bytesPerLine + i*dx;
          caddr_t dstEnd = dst + dy*bytesPerLine;
          do {
            memcpy(dst, src, dx);
            src += bytesPerLine;
            dst += bytesPerLine;
          } while (dst != dstEnd);
        }
      }
    }
    frame->Data() = newData;
  }
  return VsFilter::Work();
}

VsEntity*
VsPuzzle::Creator(Tcl_Interp* in, VsEntity* pr, const char* nm) {
  return new VsPuzzle(in, pr, nm);
}

VsSymbol* VsPuzzle::classSymbol;

void
VsPuzzle::InitInterp(Tcl_Interp* in) {
  classSymbol = InitClass(in, Creator, "VsPuzzle", "VsFilter");
}
```

E.3 The `vsPuzzle.tcl` Module Script File

```
VsPuzzle classProc panel {w orient args} {
  apply Viewport $w \
    -height 200 \
    -allowVert true \
    $args

  Form $w.form

  Label $w.form.label \
    -label "Puzzle" \
    -borderWidth 0

  VsLabeledScrollbar $w.form.dimension \
    -label "Dimension" \
    -value [$self dimension] \
    -converter "vsRoundingLinearConverter 3 10" \
    -inverter "vsLinearInverter 3 10" \
    -continuous [true] \
    -callback "$self dimension" \
    -fromVert $w.form.label

  VsLabeledScrollbar $w.form.timeStep \
    -label "TimeStep" \
    -value [$self timeStep] \
    -converter "vsLinearConverter 0 1" \
    -inverter "vsLinearInverter 0 1" \
    -continuous [true] \
```

```
    -callback "$self timeStep" \
    -fromVert $w.form.dimension

  Command $w.form.scramble \
    -label "Scramble" \
    -callback "$self scramble" \
    -fromVert $w.form.timeStep

  Command $w.form.solve \
    -label "Solve" \
    -callback "$self solve 1" \
    -fromVert $w.form.timeStep \
    -fromHoriz $w.form.scramble
}
```

E.4 The `vspuzzle` Application Script

```
proc Puzzle {w m args} {
  set dimension [keyarg -dimension $args 4]
  set scale [keyarg -scale $args 2]
  set args [keyargs {-dimension -scale} $args exclude]

  apply Form $w \
    $args
  VsScreen $w.screen \
    -scale $scale \
    -resizable true
  Command $w.scramble \
    -label "Scramble" \
    -callback "$m.puzzle scramble" \
    -fromVert $w.screen
  Command $w.dismiss \
    -label "Dismiss" \
    -callback "catch {vs destroy}; exit" \
    -fromVert $w.scramble
  Command $w.controlPanel \
    -label "Control Panel" \
    -callback "VsPanelShell $w.controlPanel.shell -obj $m" \
    -fromVert $w.scramble \
    -fromHoriz $w.dismiss
  $w.screen overrideTranslations "<BtnDown>: tcl($m position)"

  VsEntity $m
  $m set w $w
  $m proc position {} {
    set clickpos [%event position]
    set width [$w.screen getValues -width]
    set height [$w.screen getValues -height]
    set x [expr {[lindex $clickpos 0]*[$self.puzzle dimension]/$width}]
    set y [expr {[lindex $clickpos 1]*[$self.puzzle dimension]/$height}]
    $self.puzzle position [list $x $y]
  }
  VsSunVfcSource $m.source \
    -scale $scale
  VsPuzzle $m.puzzle \
    -dimension $dimension \
    -input "bind $m.source.output"
  VsWindowSink $m.sink \
    -widget $w.screen \
    -input "bind $m.puzzle.output"
}

proc main {} {
  global argv errorInfo

  xt appInitialize appContext "Puzzle" argv {}

  set w [lindex $argv 0]
  set args [lrange $argv 1 end]
```

```
  $w setValues -allowShellResize true

  vs appInitialize appContext vs

  apply Puzzle $w.puzzle vs.puzzle \
    $args
  $w realize
  vs start

  while {[catch {appContext mainLoop} msg]} {
    VsErrorShell $w.err \
      -summary $msg \
      -detail $errorInfo
  }
}

main
```

Bibliography

[1] C. J. Lindblad, D. J. Wetherall, D. L. Tennenhouse, "The VuSystem: A Programming System for Visual Processing of Digital Video," *Proceedings of ACM Multimedia 94*, October 1994.

[2] C. J. Lindblad, D. J. Wetherall, W. F. Stasior, J. F. Adam, H. H. Houh, M. Ismert, D. R. Bacher, B. M. Phillips, D. L. Tennenhouse, "ViewStation Applications: Intelligent Video Processing Over a Broadband Local Area Network," *Proceedings of the 1994 USENIX Symposium on High Speed Networking*, August 1994.

[3] D. Bacher, "Content-Based Indexing of Captioned Video," SB Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1994.

[4] D. L. Tennenhouse, J. Adam, D. Carver, H. Houh, M. Ismert, C. Lindblad, W. Stasior, D. Weatherall, D. Bacher, and T. Chang, "A Software-Oriented Approach to the Design of Media Processing Environments," *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[5] J. F. Adam, H. H. Houh, M. Ismert, and D. L. Tennenhouse, "A Network Architecture for Distributed Multimedia Systems," *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[6] B. Phillips, "A Distributed Programming System for Media Applications," SM Thesis Proposal, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1994.

[7] D. Wetherall, "A Visual Programming System for Media Computation," SM Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1994.

[8] W. Stasior, "Visual Processing for Seamless Interactive Computing," *The ViewStation Collected Papers*, MIT/LCS/TR 590, MIT Laboratory for Computer Science, Cambridge, MA, November 1993.

[9] J. F. Adam, "The Vidboard: A Video Capture and Processing Peripheral for a Distributed Multimedia System," *Proceedings of the ACM Multimedia Conference*, August 1993.

[10] T. M. Levergood, A. C. Payne, J. Gettys, G. W. Treese, and L. C. Stewart, "AudioFile: A Network-Transparent System for Distributed Audio Applications," *Proceedings of the USENIX Summer Conference*, June 1993

[11] Apple Computer Inc., "Hypercard (Version 2.2)," Apple Computer Inc., 1993.

[12] Apple Computer Inc., "Inside Macintosh: Quicktime, Inside Macintosh: Quicktime Components," Addison Wesley, 1993.

[13] SMPTE Task Force on Headers/Descriptors, "SMPTE Header/Descriptor Task Force: Final Report," *SMPTE Journal*, 101:6(411-429), June 1992

[14] "Extensions to the IEEE Standard Portable Operating System Interface for Computer Environments for the Support of Realtime Applications," IEEE Std 1003.4-1993.

[15] N. Abramson and W. Bender, "Context-Sensitive Multimedia," Proceedings of the SPIE, Vol. 1785, September 1992.

[16] ISO/IEC JTC1/SC2/W10, "Digital Compression and Coding of Continuous-Tone Still Images," IEC Draft International Standard 10918-1, 1992.

[17] Microsoft Corporation, "Microsoft Video For Windows Users Guide," Microsoft Corporation, Redmond, WA, 1992.

[18] J. K. Ousterhout, "Tk: An X11 Toolkit Based on the Tcl Language," Computer Science Division (EECS), University of California, Berkeley, CA, January 1991.

[19] D. P. Anderson, P. Chan, "Comet: A Toolkit for Multiuser Audio/Video Applications," Computer Science Division (EECS), University of California, Berkeley, CA, October 1991.

[20] J. Escobar, D. Deutsch, C. Partridge, "A Multi-Service Flow Synchronization Protocol," BBN Systems and Technologies Division, Cambridge, MA, March 1991.

[21] ISO/IEC JTC1/SC29, "Coded Representation of Picture, Audio, and Multimedia/Hypermedia Information," Committee Draft of Standard ISO/IEC 11172, 1991.

[22] J. Rees and W. Clinger, eds., "The Revised↑4 Report on the Algorithmic Language Scheme," *Lisp Pointers*, 4(3), ACM, July-September 1991.

[23] R. G. Herrtwich, "Time Capsules: An Abstraction for Access to Continuous-Media Data," *Proceedings of the 11th Real-Time Systems Symposium*, IEEE Computer Society, 11-20, December 1990.

[24] D. P. Anderson, R. Govindan, and G. Homsy, "Abstractions For Continuous Media In A Network Window System," *Technical Report No. UCB/CSD 90/596*, Computer Science Division (EECS), University of California, Berkeley, CA, September 1990.

[25] A. Hopper, "Pandora - an Experimental System for Multimedia Applications," *Operating Systems Review*, 24(2):19-34, April 1990.

[26] J. K. Ousterhout, "Tcl: An Embedded Command Language," Computer Science Division (EECS), University of California, Berkeley, CA, January 1990.

[27] D. L. Mills, "On the Accuracy and Stability of Clocks Synchronized by the Network Time Protocol in the Internet System," *ACM Computer Communication Review*, 20(1):65-75, January 1990.

[28] G. L. Steel Jr., "Common Lisp the Language," Digital Press, 1990.

[29] C. Williams and J. Rasure, "A visual language for image processing," *IEEE Computer Society Workshop on Visual Languages*, Skokie, Illinois, 1990.

[30] P. J. Asenta and R. R. Swick, "X Window System Toolkit: The Complete Programmer's Guide and Specification," Digital Press, 1990.

[31] C. Upson, T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, A. van Dam, "The Application Visualization System: A computational environment for scientific visualization," *IEEE Computer Graphics and Applications*, 30-42, July 1989.

[32] J. F. Bartlett, "Scheme->C: a Portable Scheme-to-C Compiler," Digital WRL Research Report 89/1, January 1989

[33] W. Bender and P. Chesnais, "Network Plus," Proceedings of the SPIE Electronic Image Devices and Systems Symposium, 900:81-86, Los Angeles, CA, January 1988.

[34] C. A. Csuri, S. Dyer, J. Faust, and R. Marshall, "A Flexible Integrated Graphics Environment for Supercomputers and Workstations," *Science and Engineering on Cray Supercomputers*, Cray Research, Inc., 533-548, 1987.

[35] D. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, 63(8):1897-1910, October 1984.

[36] A. Black, "An Asymmetric Stream Communication System," *Operating Systems Review*, 17(5):4-10, October 1983.

Index