Display Manager for a Video Processing System

by

Alan Wayne Blount

SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE
DEGREE OF
BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1991

Author: _____

Department of Electrical Engineering and Computer Science
May 20, 1990

Certified by: _____

V. Michael Bove, Jr.
Assistant Professor, MIT Media Laboratory
Thesis Supervisor

Accepted by: _____

Leonard A. Gould
Chairman, Departmental Committee on Undergraduate Theses

Display Manager for a Video Processing System

by

Alan Wayne Blount

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 1991 in partial fulfillment of the
requirements for the Degree of
Bachelor of Science in Computer Science and Engineering

ABSTRACT
A simple window manager for Cheops, an open architecture digital television system, is presented. The manager handles the interactive display of mulitple windows contiaining video sequences and dynamic text, and manages the allocation of resources within Cheops. The manager is optimized to support Network Plus, an application that enhances television network news by finding and displaying related wire service news articles.

Thesis Advisor: V. Michael Bove, Jr.
Assistant Professor, MIT Media Laboratory

Contents

Chapter 1

Introduction

We can do the innuendo, we can dance and sing

When it's all said and done we haven't showed you a thing

– Don Henley

Digital television, as exhorted in Media Lab lectures and demonstrations, is not necessarily about better pictures. Digital video's primary characteristic will be its flexibility. Research at the MIT Media Laboratory has resulted in a scalable representation for video signals [BL91], so that a server may deliver the requested information through the given channel, whether the channel be 1.5 Mbits/second or 20 Mbits/second, through network cable, compact disk, or aerial broadcast.

To focus on signal processing issues, however, is to see with tunnel vision. One must look wider, to new applications. After all, the ability to generate video of

varying resolution will not by itself sell a system; there must be content, preferably content unrealizable with analog television systems. Applications that merge video with text, sound, and still photography, "multimedia" applications, to use a bad word, are candidates for digital video's strongest selling point. One such potential application is described below, and a platform to support it is the topic of this thesis.

1.1 Network Plus

In 1988 Bender and Chesnais [BC88] presented a video application that makes innovative use of the digital domain. Television news, while immediate, with color and motion, lacks the depth of a good newspaper. This is less the fault of the broadcasters than of the medium itself, analog broadcast television. A speaking newscaster can only say so many words in a half hour, and must cover a broad range of topics.

A newspaper, on the other hand, while lacking television's motion and funny sportscasters, may convey far more verbal information, which allows writers to speak at greater depth about their topics. The conventions of print journalisim are also different than those of television. For instance, there is a well-defined place for explicit editorial comment.

Network Plus is an attempt at merging the two. Network Plus retains the visual format of television news, but adds newspaper articles, printed directly on the screen beside a window containing the news program. The article displayed is related to the

story being discussed on the video. The relationship is determined by pre-processing the day's articles taken from the news wire, so that they may be easily searched for key words. As a video news story is displayed, the closed-captioned text accompanying the news program is compared against the list of key words from each newspaper article. If there are greater than a threshold number of matches, the article is displayed. While this is not an ideal method of comparing the content of the two sources, it was determined to be a functioning heuristic that gets good results most of the time.

Network Plus did not end with the broadcast. In line with Electronic Publishing's larger goals [LB87], the software attempted to incorporate stills from the broadcast into a printed version of the day's news. Appropriate stills were determined through a combination of speech emphasis detection and visual scene change. No provision was made to record the entire video sequence, however, as this was before the days of inexpensive, real-time video compression. Also, the system was entirely passive—the viewer could not pause the video to finish reading an article.

The text, video, and captured stills were displayed in several non-overlapping windows on a video display. The text was displayed using Soft Fonts [Sch83], typefaces that make use of softened edges to increase readability.

1.2 Cheops

Cheops is the third in a series of video display systems developed by the Entertainment and Information Systems Group of the MIT Media Laboratory. Its predecessors, Phoenix [Wat87] and Iranscan [Wat88], are functional vector quantitization decoders, but lack Cheops' flexibility, I/O bandwidth, and processing power. Iranscan displays a video sequence with 1/4 NTSC spatial resolution in a window environment at 30 frames per second, more than sufficient for pleasurable viewing. Unfortunately, neither system has the capability to digitize or compress video. Video is captured frame-by-frame on a dedicated digitizer and encoded on a general purpose workstation. The process is tedious, as the source video must be played from a device that had a stable freeze frame, such as a VPR-3 one-inch tape deck, through an RGB encoder to the digitizer, and inevitably involves re-cabling something to correct a ground-loop or whatnot. The raw digitized source, at 20 megabytes a second, quickly fills even the largest storage devices. Finally, while video compression may be performed on a general purpose machine, dedicated hardware will do the job much more quickly.

Iranscan's greatest limitation, however, was that it was limited to playing sequences that could fit in the Macintosh's RAM, about eight megabytes, or around ten seconds of video. Ten seconds is long enough to exhibit the artifacts of different video compression schemes, but is useful for little else.

Cheops promises to solve these dilemmas. Cheops is an open-architecture video

platform for use in digital video experimentation [BL91, WB90]. A Cheops system consists of a number of input, output, and processor cards attached to a backplane. Cheops uses two internal data channels, the Global Bus and the Nile Bus. The Nile Bus is optimized for high speed video data transfers, while the Global Bus is a general purpose bus for system control. A minimal Cheops system consists of a processor card and an input or output card. Cards may be added at will, making the processing power and I/O flexible. Full size and frame rate NTSC video may be digitized, compressed on the fly, and dumped through the processor card's SCSI port to storage on a host machine's filesystem, as well as read back, decompressed, and displayed. The length of a video sequence is limited only by the compression algorithm used and the amount of storage space.

For the first time, the lab will soon be able to conduct research that utilizes long sequences of video. Adaptive compression schemes, digital editing systems, and applications such as Network Plus all require long sequences of video for development, demonstration, and use.

The bulk of processing on Cheops is currently performed on its Intel 80960CA Microcontroller. The 80960 is a 32 bit RISC processor capable of sustaining 2 instructions per clock, or 80 million instructions per second with a 40Mhz clock. Though the 80960 is supposed to be only a "Traffic Cop" that supervises DMA transfers and image compression hardware elsewhere on Cheops, to date such hardware has not

been made functional. Fortunately, the 80960 is powerful enough to display graphics and small movies by itself.

1.3 Network Plus on Cheops

As mentioned above, Network Plus has been done before. Unfortunately, it could not be left running forever, as it occupied too many machines, displays, and digitizers. It was kept running for a few months to complete debugging and to record demonstrations, and was taken down. The intent of my work is to make Network Plus and similar applications into research tools, rather than demos. The determinant is convenience. Tools that require a big setup are unlikely to be used by anyone besides their creator.

Cheops solves the resources problem. With a processor, input, and output card, a single Cheops system on a host with reasonable processing power can perform the entire news gathering, digitization, correlation, and display task.

But Cheops comes bare. The only tools provided were a processor that runs compiled 'C,' a SCSI port, and a word-addressed 1200 by 1000 line display. No "operating system" was present, besides a simple ROM monitor that was brought up with the processor card. Some sort of display manager was required. It was not the author's intention to replicate 'X,' rather, something simple was desired, that would enable the easy display of text-filled windows, and stay out of the way of the moving

video. More will be said about the text windows system in the Managers chapter.

A display system, however, requires underlying software. It is necessary to be able to run several processes simultaneously, as well as give Cheops processes a clean means of communication with the host. The following chapter describes Cheops' multiprocessing and communications systems.

Chapter 2

The Kernel

The kernel is set of procedures that comprise Cheops' operating system. It resides entirely in an EPROM on the processor card. Its responsibilities include booting the machine and providing interrupt and fault handlers for the 80960. The kernel also contains a minimal multiprocessing system, with a set of utilities that enable communication between processes on Cheops and the host.

The following sections describe the multiprocessing and communications facilities.

2.1 The Multiprocessing System

Although Cheops is not a general-purpose computer, much of the functionality of a general-purpose machine is required for Cheops to support a reasonable software base. As there are likely to be more Cheops users than Cheops machines, a time

sharing system is necessary. Also, individual users often desire to run several unrelated processes simultaneously.

Unfortunately, despite the 80960's transparent parallelism, each processor may interpret only one instruction stream at a time. Without a process swapper, the number of programs Cheops may run at once is at best limited by the number of processor cards.

What is needed is the usual sort of time sharing system, one that gives each running process an occasional slice of the processor's time. Most modern, large processors have features to support such a system. Virtual addressing is desired, for then each process may have its own address space, and the hardware can guarantee that processes do not interfere with each other's memory. Unfortunately, the 80960CA is optimized for speed and does not support virtual memory addressing. If more than one program is to run simultaneously, they must share the same address space. This is not a problem if we assume that each user program cooperates by utilizing only the space allocated to it. It will be impossible, however, to give warning to a process that has exceeded the bounds of its memory space.

So, what kind of process swapper should we implement? Cheops was designed to run real-time video compression, decompression, and management software. Most of these tasks are time-critical. A program that plays a movie should not be interrupted for a significant length of time, or the viewer will notice the halt. Hence a UNIX–like

operating system, with its arbitrary preemptive process swap times, is not feasible. What is needed is a simple process swapper that gives the user's processes control of the swap times.

2.1.1 Design

Ra, the Cheops multiprocessing system, allows up to a fixed number of processes to be resident on Cheops simultaneously. They are written to a number of fixed-length spaces in memory. When a process completes or is otherwise removed, its space is freed and another process may be downloaded into that space.

Ra gives the user control of process swapping at the procedure level with a `proc_sleep()` command. When a process has no urgent task to complete (such as between displaying frames of a movie, or while waiting for a command) it may call `proc_sleep()`.

Before the function of `proc_sleep()` is described the notion of a process's status must be introduced. A process may have one of five possible statuses, described below.

P_NULL This slot in the process table is empty.

P_HALTED Process exists, but is not being executed. Process swapper ignores this process.

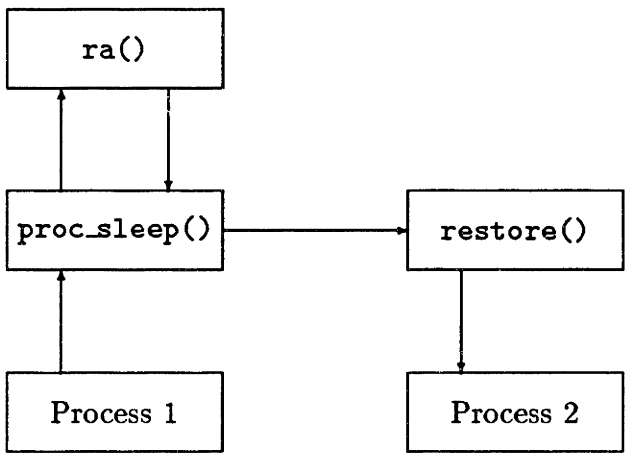

Figure 2.1: A process swap, initiated by a call to `proc_sleep()`. Up arrows indicate calls, down arrows returns.

P_ASLEEP This process is not currently being executed, but the process swapper will return control to it when its turn comes around.

P_ACTIVE Process is currently being executed.

P_NAILED This process has been killed, but is still functioning because it has not executed a `proc_sleep()`. When it does so, it will not return.

`proc_sleep()` works in three stages. Its first code segment is an assembly language procedure that saves the process' state. The process' state consists of the contents of the global and local registers, including the stack, frame, and previous frame pointers, plus the arithmetic control, process control, and special function registers. `proc_sleep()` then sets the process' status to P_ASLEEP, and calls `ra()`,

the process swapper. `ra()` finds another sleeping process (which may be the process just swapped out) and calls `restore()`, which replaces the process' environment and returns it control.

`ra()` uses a round-robin queueing algorithm to determine which process goes next. All processes have the same priority level. A more complicated process swapper could allocate processor time based on varied priority levels. While the design and implementation of a priority based swapper would be simple, it was deemed unnecessary. Processes that require frequent processing can be scheduled on interrupts, which already have a priority scheme.

The multiprocessing system uses a pair of data structures to store information about all the processes resident on Cheops. They are the proc_state structure and the usr_proc_table. Proc_state contains the number of resident processes, the process ID of the currently running process, and is also a catch-all for miscellaneous system data and flags. The usr_proc_table contains a copy of each process' state.

The kernel contains a suite of procedures that act on the above data structures to control process execution. They are as follows:

`init_multitasking()` Called by the kernel during boot. Sets up the proc_state structure and usr_proc_table.

`proc_new_local()` Places a new process in the usr_proc_table, initializing its state. Inserts a string containing the process' name in the local_name_table, which is

accessed through the proc_state structure.

`proc_new_host()` Places a string containing the name of a procedure that runs on the host in the host_name_table, which is accessed through the proc_state structure. Cheops keeps track of any processes with which it is communicating through this table.

`proc_halt()` Sets a process's execution state to P_HALTED. This process is passed over by the process swapper, but its state remains intact.

`proc_continue()` Returns a P_HALTED process to the P_ASLEEP execution state.

`proc_kill()` Removes a process from Cheops. Sets its execution state to P_NULL.

The above procedures may be accessed through the 80960's system procedure table. The sys_proc_table allows a level of indirection between a procedure's identity and its address. A set of stubs for these procedures exists in Cheops' standard library, each of which uses a "calls" instruction to reach the actual procedure in the kernel. Hence any procedure may use these functions, without the necessity of linking directly to the kernel.

Currently only a handful of applications need call the `proc_` procedures. The serial monitor is one. The monitor is the first procedure to be entered into the usr_proc_table when Cheops is booted. It functions as a typical ROM monitor, allowing the user

to display and modify registers and memory, and gives serial line control of the multiprocessing system. The user's interface to the above procedures is on a command line over the serial port.

As the SCSI communication system is made robust, however, the ROM monitor will be required less. A second set of procedures, executable from the host, use the kernel's SCSI communication system to control the multiprocessing system. More will be said about this means of communication in the Communications section.

2.1.2 Performance

Efficiency is important in a process swapper. Though no quantitative tests have been conducted on Ra, it seems to perform well if called at up to tens to hundreds of times per second, without the system taking a serious performance hit. This may be improved, as no effort was made in `sleep()` or `restore()` to take advantage of the 80960CA's multiple execution units.

There is, however, a bottleneck. Any time the 80960's Return Instruction Pointer is modified, the register cache must be flushed. This requires approximately 200 cycles, which is near the time that the rest of the swap requires.

2.2 The Communications System

The design of the Cheops-host communication system is arguably Cheops' most important software design task, for it determines how all future software will interact with the system. A transparent, flexible, intuitive package will be of immeasurable benefit to future software authors, and Cheops will likely see a lot more and better software as a result. A lousy interface will do the opposite, will likely have to be rewritten, which will make all running software obsolete.

Why is communication software required? Unlike previous specialized digital display hardware, Cheops does more than just display. It will be used as both an input and output device, as well as a processing engine. Most general purpose computers are connected to a network, are running UNIX or similar operating system, and have easy access to mass storage, remote logins, etc. Cheops is limited to two interfaces: Serial I/O at 9600 baud and host-initiated SCSI transactions at 5 to 10 Mbits/second. Serial is used only as a development and debugging tool—no applications software will use this (though Cheops' hardware designer may disagree).

The base SCSI software configures Cheops to look like a disk drive to the host. When the host transmits data to Cheops, it simply writes a number of 512 byte blocks of data into a specified region of Cheops' memory. The write is transparent—processes running on Cheops know nothing of it. Communication may be initiated only by the host. On top of this we must write a "user friendly" bidirectional communication

system.

Neither Cheops users nor programmers are interested in the SCSI connection. They would like to write or execute code that uses software running on Cheops. Further, programmers who develop Cheops executables are also not interested in the intricacies of the SCSI connection. What is needed is a facility to manage communications between the host and Cheops, as well as between separate Cheops processes running simultaneously. The following sections describe first the structure of the communication system, its interaction with the process manager, and finally how to use the system.

The design is the result of a joint effort by the author and Shawn Becker, a Ph.D. student. Becker implemented most of the message passing procedures and their underlying support code, while the author participated in their debugging and integration into Cheops' kernel. Greg Hewlett implemented the circular queue software, as well as the message passing stubs.

2.2.1 Structure

As mentioned above, the hardware provides a means of transferring 512 byte blocks of data bidirectionally between the host and Cheops, but only at the host's initiation. Several layers of procedures have been created to provide a functional interface to the transfer mechanism.

The lowest layers that we need be concerned with are the `ch_read_block()` and `ch_write_block()` procedures. Both are called from the host, and read or write a number of 512 byte data blocks when passed Cheops' file descriptor and appropriate addresses. The file descriptor is obtained by a call to `ch_open()`.

These procedures are rather unwieldy, as the buffer to be sent or read must be a multiple of 512 bytes in length. Most data communication makes use of `ch_read()` and `ch_write()`, which use the above procedures to provide arbitrary-length transfers.

On top of these are the IPC procedures. Communications between Cheops and the host are typically accomplished with the use of ports. A process that wishes to communicate with another may create a port with a `port_create()` call. The arguments to `port_create()` are the process identifiers of the source process and the destination process. Note that a port is *bidirectional*, the tag *source* denotes nothing more than which process created the port.

There are two other port management procedures. `port_destroy()` frees a port for reuse. `port_pending()` takes the name of a process and returns the port number of any port that has a message pending for that process.

The `msg_` set of procedures make use of the port system to implement data transfers. `msg_send()` and `msg_receive()` take the process identifier of the process making the call, a port number, and a pointer to a data buffer which is either transferred or filled with a transfer. Note that `msg_receive()` executes immediately. No check is

made to assure that there is valid data to be read. To make a guaranteed transfer, another layer is needed. The `msg_rpc()` call writes a block of data with `msg_send()`, repeatedly checks `msg_pending()` until there is data available, and reads this data into a local buffer with `msg_receive()`. This is the most convenient means of communication for applications writers. It enables a process running on the host to make a remote procedure call on Cheops, pass the arguments in the outgoing buffer, and return the result in the incoming buffer. The same may be done from Cheops to the host, between processes on Cheops, or even between processes on the host. Note that the message library is implemented with circular buffers, so that multiple writes may stack up and be read without havoc reigning.

Finally, a word about naming conventions. The host stubs for the `port_` and `msg_` series are prefixed with "`ch_`," to differentiate them from their counterparts that run on Cheops. In retrospect, with better library management this distinction would probably not be necessary, but it does little harm.

2.2.2 Process Management

Processes may be started on Cheops by two methods. The historic method is to use the download command over the serial monitor. This is both slow and resistant to automation, however robust. But by using the SCSI message passing system described in the above section, both the speed and ease of starting processes may be increased.

Within the kernel's SCSI handler there is a stub dispatch routine that acts on remote calls to the kernel through SCSI. The entire set of `proc_` calls may be accessed through this dispatcher. In the host's library, there is a corresponding set of stubs that access this handler. Hence, by simply linking a host procedure to this library, that procedure may call `ch_proc_start()` to initialize a process in Cheops' usr_proc_table, or `ch_proc_kill()` to remove that process, or any of the other `proc_` calls.

A higher level process in the host library makes use of these functions to start a process running on Cheops. The procedure is called `ch_download_program()`, in keeping with our naming convention. Its arguments are merely Cheops' file descriptor, acquired with `ch_open()`, and the path name of an 80960 executable file. It first checks the file system to see that the file exists. If so, it calls `ch_proc_new_local()`, which, through an RPC call, allocates space in Cheops' usr_proc_table and places the filename of the executable in a table indexed by its process identifier. The executable is then copied to the /tmp directory, and is relocated to the address returned by `ch_proc_new_local()`. It is read into the host's memory and transferred to Cheops with a call to `ch_write()`. This leaves the procedure ready to run, but in the P_HALTED state so that it is not immediately executed.

There exists a program executable from the shell called "chex" that takes a filename on the command line, downloads the file, and calls `ch_proc_continue()` to start the process running. This would seem to represent the ultimate in process execution

ease, but we can do still better, as the following section details.

2.2.3 Using the Message Passing System

The bulk of Cheops software can be placed into two classifications. There are processes that are designed to run once and exit, such as initializers, screen clearers, and test software. There are also servers, which run for long periods of time and may interact with a number of separate procedures and users on the host. Examples include movie players and text window servers, both described in greater detail in the next chapter.

Processes that run once and exit may simply be started with "chex," or with a stub that performs the download and execute with the procedures in the library. If the process and its stub must communicate, the stub calls `ch_proc_new_host()` (this is automatically called at download time for the Cheops process). This lets the kernel know that there is a new process that wishes to communicate with a process on Cheops. The stub's process identifier is returned. Then either side may create ports and send messages at will.

Servers are more complicated. Typically a server will manage a suite of procedures that may be run on command by remote calls. These procedures, however, are not somehow magically executed upon receipt of a remote procedure call. Each server must contain a process that waits for incoming commands and dispatches to appropriate command handlers. This dispatcher may be similar to the one described

in Process Management. A full example of one will be given in Section 3.2. If state is to be maintained between calls, the server is responsible for mapping the client's port number to its data. The syntax of the incoming commands is entirely up to the application developer. Each process that is to be remotely called may be given an integer identifier, for example, that is transferred to signal a call. A set of software has been developed for convenience. The generic.c library contains functions that enable one to pack an arbitrary set of shorts, longs, strings, and floats into a single buffer for easy transfer. The arguments to a procedure call, for example, may be packed into a buffer on the host, transferred to Cheops, and unpacked. While this method works, it is less than elegant, for a procedure call must be made for each argument to be packed. It is cleaner to use `sprintf()` and `scanstring()` to accomplish the same.

If a large amount of data need be transferred, the transfer should be made in a manner similar to when downloading a program: The client calls to request a transfer, giving the amount of memory required as a parameter. The server allocates the space with a `challoc()` call and returns the start address to the client. The client may then transfer the data with a `ch_write()`, and notify the server when finished, if necessary.

Writing the dispatch routines is tedious. It certainly seems possible to automate the process. One can envision a program that takes a file of procedures and creates a dispatch procedure that correctly interprets procedure names and handles argu-

ments. The difference between the handler and the procedures themselves is really only syntactic. A method for transferring large amounts of data could also be standardized. Such a program may be written if it is deemed likely that enough servers will be written to justify the effort. Until then, hand-coding the dispatch routine is sufficient.

Chapter 3

Managers

The responsibilities of Network Plus's display system, or a display system for various other interactive information services, are easily broken into two tasks. The display must present two fundamentally different types of data: video sequences and scrolling windows of text that accompany the video.

3.1 Text Windows

The Cheops text window manager makes use of the multiprocessing and communications facilities to give a process operating on the host control of text windows on Cheops. As in the original Network Plus, this text manager supports non-overlapping windows. The following section gives the details of the window system's design.

3.1.1 Design Specification

The text window manager is a straightforward design using the client–server model. The windows are managed by a suite of procedures contained in a library on the host that may be linked to any process which runs on the host. The windows contain a number of lines of anti-aliased proportionally spaced text. The windows may be of any size and color, and display any datfont available. The windows are bordered by solid rectangular borders of any size or color. Both the font and the window size are set when the window is created and may not be modified. Each Window maintains an internal ASCII representation of its text, so that if the window's display is overwritten, it may be refreshed.

The window system is accessed from the host with the following set of procedures:

`tw_init()` Must be called to use the font windows library. Downloads and executes the text window server on Cheops if it is not already running. The argument passed `tw_init()` is an empty structure that will be filled with the identity of the server, client, and port number.

`tw_create()` When provided with a set of window parameters, such as size, location, color, and the filename of a font, this procedure creates the requested window and returns a numeric identifier. If the requested font is not already resident on Cheops, `tw_create()` will download the font.

`tw_print()` Prints a string of text to the window whose identifier is given. If the string would overrun the right edge of the window, a carriage return and linefeed is inserted. If the string would overrun the bottom of the window, the text scrolls upward.

`tw_refresh()` Redraws a window and its contents.

`tw_bgcolor()`, `tw_bdcolor()`, `tw_fcolor()` Modify background, border, and font color.

Note that the `tw_refresh()` command allows windows to overlap, for windows underneath others may be redrawn at will to "bring them to the top." There is no explicit code that keeps track of which window overlaps which, however, and any material on the display that is overwritten by a text window is lost. This follows the minimalist philosophy of window systems design: each process is responsible for its own refreshing, and determining the need to refresh is left to the applications programmer. While the window system is certainly no 'X,' it will efficiently display scrolling screenfuls of text, simultaneously with video playing in another window, which is all that is required for Network Plus, electronic newspapers, and many other Cheops applications.

3.1.2 Implementation

A means of placing text characters on the screen was accomplished by porting the Garden's datfont utilities to Cheops. While there is little that is conceptually difficult within the utilities, they function in a manner unsuitable for execution on Cheops. Most critical is that the fonts are loaded from datfiles, while Cheops has no direct means to read from a filesystem. The font library's initialization command `fnt_init()`, as ported to Cheops, takes a pointer to a structure containing the font data in its "raw" form, a collection of structures, tables, and bitmaps, and generates a structure that may be passed to any procedure which uses a given font. The advantage of maintaining a "raw" structure is that, since it is never modified, a number of font structures may be generated from it, each with its own color mapping function.

The window system is the result of a straightforward implementation. The font library provides the means to display text upon any color background at any location. Each text window keeps a structure that maintains state: the current cursor location, a buffer containing an ASCII representation of the text currently being displayed, the window location, and so forth. This state is modified upon each print, refresh, or color change. Scrolling is accomplished with a primitive block copy, requiring nearly a second to scroll a block of text. It will be improved by using a DMA transfer to move each line up the screen, and should result in text that scrolls without noticeable delay.

The server works as follows: It periodically makes a `port_pending()` call to check if its services are requested. If so, it dispatches to a command parser, otherwise, it calls `proc_sleep()`, so that other processes may function. The command parser is implemented with a "switch" statement that switches on the identity of the command received, which is encoded as an integer defined in the tw.h include file.

The server makes clever use of the kernel's port based communication system to manage communication with multiple clients. Each time `tw_create()` is called, the server requests a new port and has the client send all commands associated with that window to the new port. When a command arrives on one of these active ports, the window server searches for the port number in a table that indexes window structure pointers against port numbers. Hence any number of host processes may each have any number of windows active simultaneously.

3.2 Movies

The movie player that this system requires is not yet functional. Though current movie players display sequences at reasonable frame rates, they are limited to playing uncompressed video from memory, which will only hold a few seconds of data. A movie player exists that plays uncompressed movies as they are transfered to Cheops over SCSI, but only postage stamp movies may be played at reasonable frame rates with this method.

Two additions to the hardware are required so that we may view the full length of the evening news. The first is the input card, which will digitize video in real time. Also required are the filter engines and other compression/decompression hardware. Though this hardware is not complete, the movie player has been fully specified. The simplest movie player takes uncompressed video data from the input card and transfers it over the Nile bus to the output card, where it may be displayed full frame, or in a window. This system requires almost no computation by the processor card, leaving it free to manage other display tasks, such as formatting text for Network Plus.

While direct display of video is sufficient for Network Plus, additional computation must be performed to store such video offline. Uncompressed RS-170A video requires approximately 20 megabytes per second, while the SCSI I/O bandwidth is limited to 5 megabits per second. The processor card will provide the compression, using a variety of methods which include multidimensional filtering, decimation and interpolation, discrete cosine transforms, motion estimation, signal modeling, entropy coding, scalar/vector/lattice quantitization, and additional non-linear methods [WB90].

The movie player, at the surface, is very similar to the text window server described above. It makes use of the client-server model, with a stub running on the host that manages filesystem I/O. This stub may be as simple as a "play" command that plays a given movie. Movies are stored as datfiles, whose decriptor contains

information about the encoding method used on the movie. The "play" stub will interpret this information and, after downloading and starting a movie player, will instruct the player in which decoding system to use. The stub may accept additional commands as the movie is displayed so that it may be paused, reversed, restarted, stepped frame-by-frame, and so on.

The movie player itself will work as a task scheduled on a periodic interrupt. The Cheops executible will install the interrupt and the interrupt software that actually does the decompression and block transfers to the frame buffer. The executible will remain in the process queue, and periodically check for new requests to play, pause, reverse, or halt movies. With suitable management functions, nothing prevents a single server from running as many movies simultaneously as the bus and SCSI bandwidth will allow.

Many applications, including Network Plus, require the closed captioned text that is encoded within the video signal. Since the movie player has direct access to this information, it may decode the text and make it available to other processes on a port.

A factor overlooked thus far is audio. While handling audio may not be necessary for video coming straight from the input card to the display (the audio may be played off the source), the task need be accomplished for playback of compressed video. It is likely that audio will be run on a separate machine somehow synchronized to Cheops,

possibly through the serial port. Experience has shown that synchronizing sound to video is a nontrivial problem. Synchronizing video to sound may be easier.

Chapter 4

Applications

4.1 A Simple Application

A simple application that uses the text windows package has been created. It is called "chuck," and is executable from a shell. It accepts standard input and prints it to a window on Cheops. Chuck should be useful for illustrating demonstrations with on-screen text.

While chuck does nothing that is computationally interesting, it nicely illustrates the "Cheops avoidance" method of programming on Cheops. Cheops may be used as an output device by a simple action from the host's command line. All the software and data that must be resident on Cheops to accomplish this is transparently downloaded and executed. This should serve as a model for future processes.

Cheops does what it is good at: it quickly displays text on the screen. During the design of the window system, it was questioned whether there should be an option to automatically break words. When faced with such questions, the author asked "can it be better done off line, on the host?" Given a system that simply displays text, word breaking may be done on the host, as can interpreting Postscript, or any other text task not directly related to placing characters in a window.

4.2 Network Plus

The primary intent of this thesis is to provide a bed upon which Network Plus and similar interactive display systems may be built.

Network Plus runs in three steps. First, articles are accumulated "raw" indirectly from the Associated Press newswire and other sources via Usenet and FM radio subcarrier broadcasts. Before the television news program is displayed, these articles must be preprocessed so that their content is readily discernible by the matcher. Preprocessing involves generating a list of "keywords," alphabetized so that an logarithmic search may be used. The keyword list is simply the list of words used in each story with "noise" words removed. Noise words are words such as "the," "said" and "today," [BC88] which contain no useful information. Preprocessing may be done by any machine with access to the filesystem containing the articles.

When the news is viewed, it may be displayed in a window covering perhaps a

quarter of the display. In fact, a window containing the full RS-170A resolution version of the news program displayed with the current output card in its 1024 line high resolution mode would occupy less than one quarter of the screen. The remaining three quarters may be filled with windows containing text and still frames taken from the news program or other news sources.

By using the text window and movie servers, these windows may be easily generated. Let's look at a short application that implements a mock-up of Network Plus by playing a movie and simultaneously displaying a still frame and a text window.

The code in Figure 4.1 runs on the host. The text window and movie stubs, such as `tw_init()` and `mv_play()`, are accessed by linking this code to the appropriate libraries. Note that this code makes no mention of ports, process identifiers, or other low level communications commands. These calls are all buried within the text window and movie stubs.

On Cheops, the text window and movie servers are installed by `tw_init()` and `mv_init()` and will continually check their input queues for commands such as those sent by `tw_print()` and `mv_play()`.

Actual implementation of Network Plus will probably be performed by resurrecting much of the original code and grafting a display system onto it using the procedures in the mock-up.

Figure 4.2 shows Cheops' display after running a similar program.

```
#include<stdio.h>
#include<strings.h>
#include<ch_kernel.h>
#include<tw_stubs.h>
#include<mv_stubs.h>

main()
{
     tw_struct tw;
     mv_struct mv;
     long window;
     int fd, color = 0x224400, ocard = 0; left = 600...;
     char buf[1024], font[32];

     ch_clear(color);       /* clear display */
     tw_init(&tw);          /* initialize text window system */

     /* open text window */
     sprintf(font, ''/sequence/datfonts/cs32'');
     window = tw_create(&tw, font, ocard, left, top...);

     /* print news article to window */
     fd = open("article1", O_RDONLY);
     read(fd, buf, 1024);
     tw_print(&tw, window, buf);
     close(fd);

     mv_init(&mv);                        /* initialize movie player */
     mv_play(mv, frame_name, x1, y1);     /* display still frame */
     mv_play(mv, movie_name, x2, y2);     /* play video sequence */
}
```

Figure 4.1: Network Plus mock-up program, C pseudocode

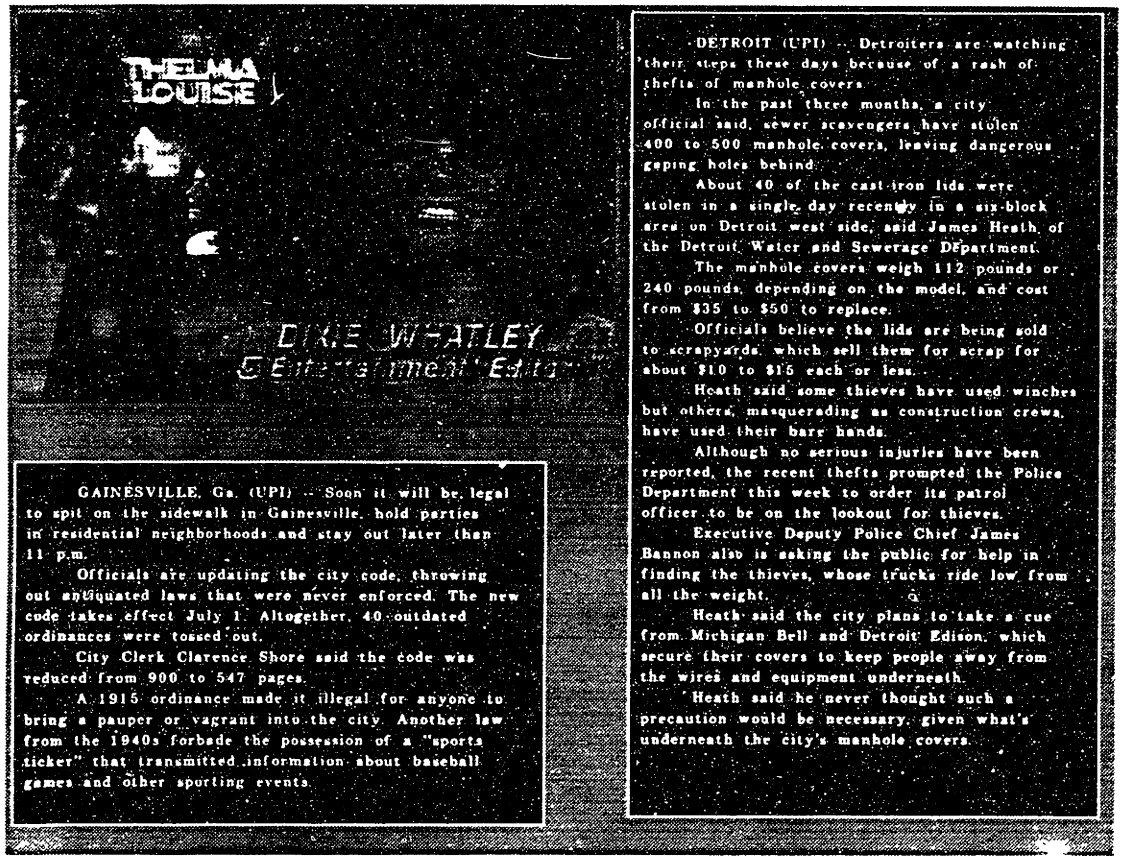

Figure 4.2: Network Plus display mock-up

Chapter 5

Conclusion

The opportunity to occasionally copy a frame from the news telecast and display it statically for several seconds may by itself increase a viewer's perception of the news, especially if the picture illustrates something found both in the news program and a related article. How can an "interesting" still frame be separated from the rest of the news? Obviously the viewer doesn't want to see stills of the newscasters at their desk. Is it possible to build a filter that will tell whether a given frame contains a news anchor or something else? One must remember that most television news broadcasts follow a rigid format—one or two fixed shots of the anchor's upper body and desk, interspersed with sequences taken on location, the weatherman at his map, and so on. It may be possible to "find the anchor" by simply examining the color statistics of a scene, or through block matching. The fixed camera and periodic nature of the scene

containing the newscasters should lend a strong statistical hand. The same software used to compress video may be used to determine useful information about a scene's content.

With higher level knowledge of the news's video content, Network Plus may be able to go well beyond its original scope. The capability to store and retrieve news broadcasts allows the source of the video data to be abstracted out. Network Plus need not know whether the video data was just digitized or is being retrieved from the host and decompressed. Hence, by recording different news programs on several input cards or by recording news programs broadcast at different times, the system may collect the day's news from several sources, and attempt to integrate these programs into a more cohesive format. By using software that segments news programs into stories, stories covering the same event may be grouped together. This may be used as an instrument to detect bias, or simply to give the viewer several angles on the same topic.

Can an algorithm be devised that determines where one story ends and another begins? Bove [Bov85] designed a system that did this in 1985 using a workstation augmented with a videocassette recorder. Information in the closed caption text is helpful. ABC News, for instance, delineates stories with a special sequence of characters. Unfortunately, the caption information is typically typed as it is spoken by the newscaster, complete with occasional mistakes and a variable delay. Hence

there is no way to determine from timing information alone which text is associated with which video segment, though the two are loosely related. Perhaps by combining timing information with scene change detection software this can be better achieved.

Television news personalization is an obvious next step. If the system can segment articles and has some knowledge of the user's interests, it may display only those articles that it thinks the user wants to see. Such is at the cutting edge of current Media Lab research.

The concept of running "display servers" on Cheops that show movies, text, and perform content detection is powerful, for it lets Cheops do the video processing for which it was designed. The rest: article matching, data management, and other filesystem oriented tasks, are best left to the machine with the filesystem, the host. The multiprocessing system and communications managers provide the bridge across this division.

Bibliography

[BC88] Walter Bender and Pascal Chesnais. Network plus. In *Proceedings, SPIE Electronic Imaging Devices and Systems Symposium*, volume 900, pages 81–86, Los Angeles, CA, January 1988.

[BL91] V. Michael Bove, Jr. and Andrew Lippman. Open architecture television. Presented at SMPTE 25th Television Converence, Febuary 1991.

[Bov85] V. Michael Bove, Jr. Personalcasting: Interactive local augmentation of television programming. Master's thesis, Massachusetts Institute of Technology, September 1985.

[LB87] Andrew Lippman and Walter Bender. News and movies in the 50 megabit living room. presented at IEEE Globecom, November 1987.

[Sch83] Chris Schmandt. Fuzzy fonts: Analog models improve digital text quality. In *Proceedings, NCGA*, pages 549–448, 1983.

[Wat87] John A. Watlington. A decoder for vector coded color motion picture sequences. Master's thesis, Massachusetts Institute of Technology, May 1987.

[Wat88] John Watlington. Iranscan movie player technical description. Technical report, MIT Media Laboratory, September 1988.

[WB90] John Watlington and V. Michael Bove, Jr. The cheops imaging system. Technical report, MIT Media Laboratory, October 1990.