A Mobile System
for Distributed Multimedia Applications

by

Patrick Wai-Ho Chan

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

© Patrick Wai-Ho Chan, MCMXCVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author ..........................................................
Department of Electrical Engineering and Computer Science
May 26, 1996

Certified by ......................................................
David Tennenhouse
Principal Research Scientist, Thesis Supervisor

Accepted by .................

Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Theses

JUN 11 1996

LIBRARIES

ARCHIVES
A Mobile System
for Distributed Multimedia Applications

by

Patrick Wai-Ho Chan

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 1996, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

This report describes the design and implementation of a mobile system for distributed multimedia applications. Most current multimedia systems rely on the computing power of desktop workstations and high bandwidth local area networks, which confine the use of multimedia applications to fixed locations in limited environments. By constructing a distributed mobile multimedia system, a new area of multimedia applications can be studied and developed.

The mobile system hardware is a PC laptop computer integrated with portable multimedia and wireless networking equipment. The VuSystem, a real time interactive multimedia programming system, is used for live media processing and programming. I developed a video capture device driver and media source/sink modules for the VuSystem to access the multimedia hardware. Performance results demonstrate that the mobile system has sufficient capability to process live media information, but reveal bottlenecks in the video capturing and wireless networking hardware. Given the system's scalable architecture, performance will improve as new technologies become available.

Thesis Supervisor: David Tennenhouse
Title: Principal Research Scientist
Acknowledgments

I would like to give my sincere thanks to my supervisor David Tennenhouse, who has supported me throughout the project. I also want to thank my colleagues for sharing the burden with me at work. Last but not least, I want to appreciate the caring and love from: Mom, Dad, Raymond, Monica, and Anita.
# Contents

1 Introduction ........................................... 11
  1.1 Motivation ........................................ 12
  1.2 Issues ........................................... 12
  1.3 Organization of this Report ....................... 13

2 Related Work ......................................... 15
  2.1 Related Work in Mobile Systems .................. 15
  2.2 Perspective ....................................... 17
  2.3 Building Blocks of the Mobile Station .......... 17
  2.4 The VuSystem ..................................... 17
  2.5 The Linux Operating System ...................... 18
  2.6 The PCMCIA Standard ............................. 19

3 Approach ............................................. 23
  3.1 Functional Components ........................... 23
  3.2 System Architecture ............................... 23

4 Processing System .................................... 27
  4.1 Hardware Components .............................. 27
  4.2 Operating System ................................ 28
  4.3 The VuSystem ..................................... 29
  4.4 Media Synchronization ............................ 33
  4.5 Modules as Interfaces to Devices ............... 34
  4.6 Performance ..................................... 34

5 Video Input Subsystem ................................ 37
  5.1 Video Capture Hardware ........................... 37
  5.2 Software Architecture ............................. 39
  5.3 CardCam Device Driver ............................ 39
  5.4 CardCam Video Source Module ..................... 46
  5.5 Performance ..................................... 51
6 Audio Input/Output Subsystem
   6.1 Design .............................................. 59
   6.2 UNIX Sound System ................................. 61
   6.3 The Linux Audio Source Module .................... 64
   6.4 The Linux Audio Sink Module ....................... 67
   6.5 Control Panel for the Linux Audio Modules .......... 69
   6.6 The AudioFile System ............................... 69
   6.7 Summary ............................................. 70

7 Wireless Network Subsystem ............................... 71
   7.1 Network Hardware .................................... 72
   7.2 Network configuration ............................... 72
   7.3 Software Architecture ............................... 73
   7.4 WaveLAN device driver ............................. 73
   7.5 VuSystem Network Modules ......................... 75
   7.6 Performance Results ................................. 77

8 Integration and Evaluation ................................ 79
   8.1 Applications ........................................ 79
   8.2 Conflicts in Integration ............................. 80
   8.3 Performance Results ................................. 81
   8.4 VuSystem Filter Modules Processing Times .......... 82
   8.5 Total System Throughput ............................ 84
   8.6 Video Transmission Throughput ..................... 86
   8.7 Video Transmission Latency and Jitter ............. 90

9 Conclusion .............................................. 93
   9.1 Summary of Work .................................... 93
   9.2 Lessons Learned .................................... 94
   9.3 Future Work ........................................ 95

A Modules In the VuSystem ................................ 97
   A.1 The VsCardCamVideoSource Module .................. 97
   A.2 The VsLinuxAudioSource Module ..................... 101
   A.3 The VsLinuxAudioSink Module ....................... 103

B The CardCam Device Driver ............................... 107
   B.1 The cardcam.h Device Driver Header File .......... 107
   B.2 The cardcam_cs.c Device Driver Source File ...... 108
   B.3 The fgrabber Startup Script File ................... 124
List of Figures

2-1 The Linux PCMCIA System Components ........................................... 20

3-1 Mobile node functional components ............................................... 24
3-2 Mobile node system architecture .................................................. 25

4-1 Mobile system picture ................................................................. 28
4-2 VuSystem application structure .................................................... 30
4-3 Module data protocol ..................................................................... 31
4-4 Edge detection program block diagram .......................................... 33
4-5 Edge detection program code fragment ........................................... 33

5-1 Video input subsystem components ............................................... 38
5-2 Miniature camera picture .............................................................. 39
5-3 Sample image capture code ............................................................ 41
5-4 CardCam PCMCIA configuration file entries .................................... 42
5-5 file operation structure ................................................................. 43
5-6 The CardCam video source module. ................................................. 46
5-7 The brightness subcommand source code ....................................... 50
5-8 CardCam video source module control panel .................................... 51
5-9 The brightness control panel source script ....................................... 52
5-10 Simple program to measure VuSystem video capture throughput ...... 54
5-11 Video capture latency in 320x240 pixel mode. ............................... 57
5-12 Video capture latency in 160x120 pixel mode. ............................... 58

6-1 Audio subsystem components ....................................................... 60
6-2 Code fragment to set the PCM volume of a mixer device. ................. 63
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-3</td>
<td>Code fragment to capture audio.</td>
<td>64</td>
</tr>
<tr>
<td>6-4</td>
<td>The Linux audio source module</td>
<td>65</td>
</tr>
<tr>
<td>6-5</td>
<td>The Linux audio sink module</td>
<td>67</td>
</tr>
<tr>
<td>7-1</td>
<td>Wireless network configuration.</td>
<td>73</td>
</tr>
<tr>
<td>7-2</td>
<td>Wireless network subsystem components</td>
<td>74</td>
</tr>
<tr>
<td>7-3</td>
<td>WaveLAN entries in PCMCIA configuration file</td>
<td>74</td>
</tr>
<tr>
<td>7-4</td>
<td>VuSystem TCP client sample program script</td>
<td>75</td>
</tr>
<tr>
<td>7-5</td>
<td>VuSystem TCP server sample program script</td>
<td>76</td>
</tr>
<tr>
<td>7-6</td>
<td>Wireless network TCP throughput measurement results</td>
<td>78</td>
</tr>
<tr>
<td>8-1</td>
<td>The Whiteboard Recorder VuSystem application</td>
<td>80</td>
</tr>
<tr>
<td>8-2</td>
<td>The Video Rover and its application user-interface.</td>
<td>81</td>
</tr>
<tr>
<td>8-3</td>
<td>Filter processing time measurement setup</td>
<td>83</td>
</tr>
<tr>
<td>8-4</td>
<td>Representative filter processing time results</td>
<td>83</td>
</tr>
<tr>
<td>8-5</td>
<td>System throughput measurement setup</td>
<td>85</td>
</tr>
<tr>
<td>8-6</td>
<td>X Window video transfer throughput measurement setup</td>
<td>86</td>
</tr>
<tr>
<td>8-7</td>
<td>TCP video transfer throughput measurement setup</td>
<td>87</td>
</tr>
<tr>
<td>8-8</td>
<td>X Window video transfer latency measurement result</td>
<td>92</td>
</tr>
<tr>
<td>8-9</td>
<td>X Window video transfer latency measurement results</td>
<td>92</td>
</tr>
<tr>
<td>9-1</td>
<td>Video Rover II media processing components</td>
<td>96</td>
</tr>
</tbody>
</table>
List of Tables

4.1 VuSystem artificial sources and sinks modules frame rate 35
5.1 CardCam device driver I/O control commands. 40
5.2 VsCardCamVideoSource module object subcommand. 49
5.3 CardCam device driver video capture frame rate. 53
5.4 VuSystem video capture frame rate 55
6.1 Mixer channels. 62
6.2 Mixer device ioctl commands. 63
6.3 Audio device ioctl commands. 65
6.4 VsLinuxAudioSource module object subcommand. 67
6.5 VsLinuxAudioSink module object subcommand. 68
7.1 Wireless network TCP throughput measurement results. 77
8.1 Representative filter processing time results 84
8.2 Representative VuSystem application throughput results 85
8.3 Video transfer with compression throughput measurement results 89
8.4 Two-way video transfer measurement results 90
Chapter 1

Introduction

Traditional multimedia systems, which rely on the computing power of desktop workstations and high-speed local area networks, confine the use of multimedia applications to fixed locations in limited environments. Mobile systems, on the other hand, focus on portability but lack multimedia capabilities. Recent advances in mobile computing and wireless network technologies suggest the construction of multimedia mobile systems for distributed multimedia applications. In this report, I describe the design and implementation of a multimedia mobile system constructed by integrating off-the-shelf wireless networking and multimedia equipment with a general-purpose PC laptop computer, and with the VuSystem [17], a user-programmable and extensible multimedia programming system. The VuSystem provides the multimedia processing software that allows the user to manipulate live media data on the mobile system.

This mobile system is unique in its integration of off-the-shelf hardware with a user-programmable and extensible multimedia system running on a general purpose operating system. The use of off-the-shelf hardware leads to low equipment cost, general availability, and the rapid deployment of new technologies. The use of a user-programmable and extensible multimedia system eases the development of multimedia applications and facilitates the adoption of new hardware. The use of a general purpose operating system facilitates software portability across hardware platforms.
1.1 Motivation

Traditional multimedia systems are used primarily for the capture, storage, retrieval and transmission of media data. Multimedia applications statically define a set of multimedia functions and only provide the delivery of stored multimedia information (e.g., video on demand, electronic encyclopedia) or simple multimedia teleconferencing between hosts [8, 26, 9]. Processing power requirements limit the set of primitive processing functions available to multimedia applications and confine their use to powerful desktop workstations with high-end video and audio equipment on high-speed local area networks.

With increasing processing power, computers can dynamically process live media data, present media information to their users, and actively interact with the users and the environments they are in. For example, using the VuSystem [17] multimedia system, multimedia applications not only deliver media data but also actively participate in manipulating and digesting the media stream. However, without the ability to move the host around together with the user, the human-machine interactions would be confined to the desktop, thus limiting the range of applications. By moving the system to a mobile station, multimedia applications can be used in a much greater variety of environments and can give closer machine-user interactions. With the advances of mobile computer technologies, new generation laptop computers provide enough power to process multimedia information. PCMCIA expansion devices add multimedia and wireless network capabilities to laptop computers.

1.2 Issues

A mobile platform for distributed multimedia applications must satisfy the following requirements:

1. **Multimedia capabilities**: The mobile station needs to have video capture and display devices, audio capture and playback devices, and a high performance processor for media computation.

2. **Mobility**: The mobile station needs to be reasonably small and light, and have wireless networking capabilities.

3. **Media processing**: The mobile station needs to have a user-programmable and extensible media programming system. It should also support the distribution of computing
resources across the network.

In building the mobile system, I designed an extensible architecture that allows new subsystems to be added to the main media processing system. The VuSystem, a multimedia programming system, is the software that runs the media processing system. Subsystems provide media input/output facilities and communicate with the VuSystem through VuSystem I/O modules. I constructed a video input subsystem, an audio input/output subsystem and a wireless network subsystem as the basic subsystems to provide media I/O and communication functionalities required by the mobile station. New subsystem can be added in the future to utilize new devices or to provide new functionalities.

In constructing the subsystems, I developed a video capture device driver and several new VuSystem source and sink modules for the VuSystem to access the video capture card and the audio input/output devices. Performance results of sample multimedia applications demonstrate that the mobile system has enough power to process live media information.

1.3 Organization of this Report

This report is divided into nine chapters. Chapter 2 summaries related work in the areas of multimedia systems, mobile computing and wireless networking, and put this research in perspective. Chapter 3 describes the approach in building this mobile system. It presents the overall architecture, which consists of the main processing system and a set of media input/output subsystems. The next four chapters describes and evaluate each subsystem in detail: Chapter 4 describes the processing system; chapter 5 describes the video input subsystem; Chapter 6 describes the audio input/output subsystem; and Chapter 7 describes the wireless network subsystem. Chapter 8 evaluates how well these systems are integrated together and describes the overall performance results. Chapter 9 concludes this report and suggests future work.
Chapter 2

Related Work

This chapter describes previous work related to the development of the mobile system. I will first describe commercial and research mobile systems to put my work in perspective. Then I will describe previous works that the mobile system are built on.

2.1 Related Work in Mobile Systems

This research is related to mobile system and wireless networking. Although general purpose laptop computer technology is advancing rapidly, there is also another class of commercial and research mobile systems that focuses on mobility, user-interface and networking capabilities by using specialized hardware and customized software. This section lists some of the related work in these areas.

2.1.1 Newton, MagicLink, Envoy and other PDAs

A Personal Digital Assistance (PDA) is a small, portable, hand-held device that assist a user in his daily work. Several vendors offered different kinds of PDAs during the past few years. Most PDAs has note-taking, scheduling, phone directory, and general application programming capabilities. Some PDAs also have hand-writing recognition, and limited networking (e-mail, fax) facilities.

PDAs primarily focus on portability and usability. Therefore, they don't offer high processing power or multimedia capabilities.
2.1.2 InfoPad

The InfoPad project [20] at the University of California (Berkeley) is developing the hardware, software and mobile network support that allow a high density of mobile users to access real-time multimedia data from high speed networks using a portable terminal called the InfoPad. The InfoPad terminal is able to send and receive audio, send pen locations from the LCD pen tablet, and receive text/graphics and full motion video. It also supports hand writing and speech recognition.

The project emphasized on building the InfoPad hardware and software as a mobile terminal with good user-interface and networking support. The InfoPad itself has no general, user-accessible computation facilities. All media computation is done at the high speed multimedia servers on the network backbone. The servers then communicate the results with the InfoPad using a picocellular wireless network.

2.1.3 PC-Notebook based Mobile Networking

The GloMo (Global Mobile) project at the University of California, Los Angeles, integrated multimedia adaptive wireless networking capabilities in a PC-Notebook platform. [12] Adaptive wireless modems and rate-adaptive video codecs are used to support multimedia applications in a wireless environment. A new architecture for the PC-notebook is suggested to overcome the limitation of the PC shared bus architecture.

The project emphasizes the hardware, low-level software and operating system support for the real-time transmission of multimedia data. Although this project is built on PC notebook and off-the-shelf technologies, the researchers have concludes that the current PC notebook technology is not adequate and a new notebook architecture is needed.

2.1.4 Wireless Overlay Networks

The Daedalus Project at the University of California (Berkeley) is investigating wireless internetwork technologies that allow mobile users to connect to a heterogeneous wireless overlay networks. [14] A mobile user can continue to operate when “roaming” from one network coverage area to another. This extend the reach of mobile systems to wide-areas using diverse wireless network technologies. A wireless data networking architecture was developed that integrate diverse wireless technologies into a seamless internetwork. Appli-
cation support services were developed to allow applications to continue to operate when roaming across different networks.

This project focuses on building a wireless networking infrastructure that allows mobile hosts to transparently access a heterogeneous wireless overlay network.

2.2 Perspective

Developments in mobile computing, wireless network technologies and multimedia systems are recent trends in the computer industry. A lot of research and development results have been commercialized and widely deployed. While much research effort focused on improving each of these areas, less effort investigated how different technologies can be integrated to build a mobile multimedia system. The mobile system described here unique in that it uses a software-intensive approach both to manipulate live video data, and also to integrate different hardware components. This approach offer flexibility in media programming and future extension of hardware components.

2.3 Building Blocks of the Mobile Station

The mobile station, for the most part, is an integration of technologies in mobile systems, multimedia system, and wireless networking. The following sections describes some of the main building blocks on which the mobile station is build: the VuSystem is the multimedia programming system running on the mobile system; Linux is the underlying operating system; the Linux kernel PCMCIA system and the PCMCIA standard provide a standard interface for the adding peripherals to the laptop, which is used for the video capture card and wireless network adaptor.

2.4 The VuSystem

The VuSystem [17] has been developed by the Telemedia, Networks and Systems Group at the MIT Laboratory for Computer Science. It is a programming system for computer-participative multimedia application, in which the application actively digest media data and provide more useful results to the user.

The VuSystem uses a media-flow architecture to process data. A media stream is pro-
cessed by connecting different data processing modules, forming a directed media processing pipeline. The VuSystem also provides an interactive programming environment for real-time manipulation of media data [27]. A distributed programming extension to the VuSystem (VuDP) [22] was developed by Brent Philips to facilitate distributed media computation in which different modules can reside on different hosts on a network.

The VuNet, an ATM-based gigabit per second local area network, was developed to provide a high-speed media transmission backbone linking workstations and custom video capture peripherals. [1]

The VuSystem can run on UNIX-based workstations not specifically designed for real-time multimedia applications.

2.5 The Linux Operating System

As part of this work, the VuSystem has been ported to Linux, which runs on the mobile system. Linux was developed by Linus Torvalds and other people within the Internet community. It is a free UNIX clone that runs on the Intel i386 series microprocessor.¹ All of its source code is available to the public which makes it ideal for development. The performance of Linux is adequate and it is quite popular among UNIXes for the PC.

The Linux support of the ELF (Executable and Linking Format) binary format and kernel loadable modules are described below in more details because they are closely related to the development of the mobile station.

The ELF binary format

ELF is a binary format originally developed by USL (UNIX System Laboratories). It is currently used in Solaris and System V Release 4. The Linux GCC and C library developers decided to move from the older a.out format to ELF [2] as the standard binary format in 1994 because ELF makes it much simpler to support shared libraries. The object files only need to be compiled to produce position-independent code and then linked to form a shard library. The shared libraries for the Linux port of the VuSystem have been produced in this

¹Linux has been ported to the Motorola 680x0 platforms and the DEC Alpha platforms. Ports to MIPS, PowerPC, and PowerMAC are under way.
manner.²

Kernel Loadable Module

Loadable modules are kernel code which are not linked directly in the kernel. They can be inserted to and removed from the running kernel at almost any time as needed. Using loadable modules reduces the size of the kernel because many kernel device drivers and components are loaded only when needed. It also makes kernel extension development much easier because loadable modules are compiled separately from the kernel and the source code can be released independently of the main kernel source.

The CardCam PCMCIA video capture device driver (described in Section 5.3) is an example of a loadable module. The device driver is loaded and integrated into the running kernel when the CardCam PCMCIA card is inserted to the system.

2.6 The PCMCIA Standard

The PCMCIA (Personal Computer Memory Card International Association) standard [19] (also known as the PC Card standard) is a platform-independent standard to handle I/O and memory expansion through expansion slots. It is used mostly in, but not limited to, laptop computers. This standard describes the mechanical, electrical and software interfaces from the PCMCIA expansion card up to client software. Introduced by the Personal Computer Memory Card International Association in September 1990, it has gone through several major revisions. The original Release 1.0 standard describes memory expansion IC cards. Release 2.0 and 2.01 specifications include I/O expansion bus support. Release 2.1 includes support for a wider range of peripherals such as modems. Several major enhancements were introduced recently. The “PC Card 95” specification released in October 1994 introduced the bus-mastering 32-bit CardBus. The Zoom Video specification introduced in March 1996 include support for high performance digital video input.

The Quadrant CardCam video capture card and the AT&T WaveLAN Wireless Ethernet Card used with the mobile station are based on the PCMCIA release 2.1 standard. While the 32-bit CardBus and the Zoom Video standard provide higher performance and more

²Because the ELF specification is not yet completed for Linux or the DEC Alpha platforms, there is no shared library in the DEC Alpha Linux port of the VuSystem.
functions, few products based on these new standards are available at this time.

2.6.1 Linux Kernel PCMCIA System

The Linux kernel PCMCIA system (also known as the PCMCIA Card Services for Linux) written by David Hinds [10] provides an interface for the operating system and the user programs to access and control PCMCIA devices. Figure 2-1 shows the components in the PCMCIA system. This system consists of three layers. At the lowest level is the Socket Services, which controls the PCMCIA socket hardware. The next level is the Card Services. Device drivers for specific cards lie on top of Card Services. Client programs access PCMCIA cards through their corresponding device drivers. For example, the VuSystem VsCardCamVideoSource module accesses the video capture card through the CardCam device driver.
Socket Services

Socket Services provides a hardware abstraction layer that allows Card Services software to utilize different PCMCIA socket controller chips. Socket Services provide an interface to initialize the socket and PC Card, setting up memory and I/O windows and handling special card events like card insertion or ejection. The Linux PCMCIA socket services consists of two loadable modules: The TCIC module supports the Databook TCIC-2 family of controllers; The i82365 module supports the Intel i82365sl family of socket controllers. One of these socket services module is be loaded depending on the socket controller hardware of the system. In the case of the IBM ThinkPad 760C used in this work, the i82365 module is selected.

Card Services

Above Socket Services is the Card Services. It provides an API for the client device drivers to access, control and configure both the card resources and the operating system resources. For example, it maps the memory and I/O ports between the card and the host, and allocates an available IRQ line for the card. It also provides extra facilities to handle memory transfers between the host and the PCMCIA card.

Client Drivers

Device drivers for specific PCMCIA cards act as Card Services clients. One special Card Services client, Driver Services, allows user-level PCMCIA utility programs to access Card Services functions and other clients. It acts as a super client and keep track of all other client drivers.

The VuSystem access the CardCam video capture card and the WaveLAN network card through the corresponding client drivers on top of Card Services. The design and implementation of the CardCam device driver is presented in Section 5.3. The design of the WaveLAN network card device driver is presented in section 7.4.
Chapter 3

Approach

The construction of the mobile system requires the integration of different hardware (multimedia and networking devices) and software (device drivers, media processing software) components. I designed an architecture that separates the media I/O hardware and software from the main media processing system. A common interface (VuSystem I/O modules) is used for the main processing system to access the media I/O devices. This allow the easy addition of new components as new technologies become available.

3.1 Functional Components

Figure 3-1 describes the functional components of the mobile system. A camera and a microphone provide the video and audio input signal. The media capture components digitize the input signals and feed them to the media processing component. The media processing component executes the multimedia program, maintains the user interface and manipulates the media data. The wireless network component provides global connectivity to the media processing component so it can transmits the processed media data to or receives additional media data from other other mobile hosts or workstations on the network. Finally, the media rendering subsystem outputs media data to the user through a loudspeaker and a monitor.

3.2 System Architecture
Figure 3-1: Functional components of the mobile node.

Figure 3-2 shows the system architecture of the mobile system. The processing system is the central processing engine for the programming and manipulation of live media data. The processing system hardware is a high-performance Pentium laptop computer. The media processing software is the VuSystem. It is ported to Linux to run on the laptop. A set of media input and output subsystems provide interfaces for the processing system to access the I/O hardware devices. This architecture allows new I/O devices to be included to the system by adding new subsystems without changing the main processing system.

A media I/O subsystem usually consists of the following parts: a media input or output device, a device driver in the kernel that presents an interface for user programs to access the device, and a VuSystem module that connect the VuSystem in-band processing stream to the device. The following are the media I/O subsystems used in the mobile station:

- **Video Input Subsystem:** The video input subsystem provides a video source for the mobile host. It uses the CardCam PCMCIA video capture card to capture live video. The CardCam device driver controls the video capture card through the PCMCIA
Figure 3-2: System architecture of the media processing components in the mobile station.
Card Services. The captured video data are then fed to the CardCam video source module, which converts them to video payloads for processing by other modules. See Chapter 5 for a detailed description of the video input subsystem.

- **Video Output Subsystem:** We use the X Window System for video display. X Window System is a network-transparent windowing and user-interface standard. It is already in use with VuSystem in other UNIX platforms. The VuSystem window sink module displays video frames on an X display. I use the window sink module without modification on the mobile host running the X Window System.

- **Audio Input and Output Subsystem:** The audio input and output subsystem provides an interface for VuSystem applications to capture and playback audio. The audio input and audio output subsystem are described together because an audio device usually offers both audio capture and playback functions and the same device driver provides both operations. The Unix Sound System (USS) device driver provides an interface for application programs to use the audio hardware on the laptop. VuSystem applications access the audio device either through the Linux audio source/sink modules, or through the AudioFile source/sink modules. The Linux audio source and sink modules directly capture and playback audio using the USS device driver. The AudioFile source and sink modules, on the other hand, use the DEC AudioFile system to record and playback audio. AudioFile is like X Window System for audio. It provides network transparency and a uniform API across different platform. It is ported to Linux to run on the mobile station.

- **Wireless Network Subsystem:** The wireless network subsystem provides an interface to connect to the wireless network. The device driver controls the wireless network adaptor and creates a regular Ethernet device interface for client programs to access the network. The Ethernet interface allows regular VuSystem network modules to access the wireless network without modification.

The processing system, the video input subsystem, the audio input/output subsystem, and the wireless network subsystem are described in more detail in the following chapters.
Chapter 4

Processing System

The processing system is responsible for the manipulation of multimedia data. It is also the multimedia application programming system, with which all other input, output and processing subsystems are built on. This chapter describes the hardware and software that constitute the processing system.

4.1 Hardware Components

Figure 4-1 shows a picture of the mobile station. Since laptop computers generally lack multimedia and wireless networking capabilities, I used PCMCIA expansion cards to add these capabilities to the mobile station. The use of PCMCIA expansion cards also decouples the laptop computer hardware from these peripherals and allows easy future upgrade of these components. The following lists the hardware components used for different multimedia networking functions:

- **Media processing**: The IBM ThinkPad 760C laptop computer [7] is chosen because of its high processing power, large display, built-in audio capability and good compatibility with Linux. The laptop includes a 90MHz Intel Pentium processor, 12.1" high-resolution LCD display, built-in audio I/O and PCMCIA support.

- **Video capture**: The Quadrant CardCam Video-IN PCMCIA video capture card is used to capture live video. It uses the Brooktree Bt812 video decoder chip for video signal decoding.
• Video display: The active-matrix color LCD screen on the laptop displays video data. It uses a Cyber 9320 video chipset (compatible with the Trident 9320LCD) with one megabyte of RAM.

• Audio capture: The IBM MWave DSP chip, in its SoundBlaster compatible mode, captures audio using the built-in microphone.

• Audio playback: The IBM MWave DSP chip, in its SoundBlaster compatible mode, plays back audio using the built-in speaker.

• Wireless network: The AT&T WaveLAN PCMCIA wireless network adaptor is used for wireless networking.

4.2 Operating System

The operating system used is Linux, a free UNIX clone that runs on the Intel i386 series microprocessor. I ported the VuSystem to Linux with shared library support.

The development of this mobile station depends on some Linux kernel features and other Linux software packages:

• ELF binary format support: The use of the ELF binary format eases the construction of shared library for the VuSystem.
• Kernel loadable module support: The CardCam PCMCIA video capture card device driver is implemented as a Linux loadable module. It is compiled separately from the kernel and inserted into the running kernel upon the insertion of the PCMCIA card. This reduces the size of the running kernel and simplifies the development of the device driver.

• PCMCIA support: Linux has very good PCMCIA support through the PCMCIA Card Services developed by David Hinds. The CardCam video capture device driver and the WaveLAN Ethernet device driver used in the mobile station depend on PCMCIA Card Services to access the device.

• Sound support: The Unix Sound System developed by Hannu Savolainen provides an universal API to access various PC audio hardware configurations.

• X Window System: XFree86 provides X Window support for VuSystem. XFree86 is a free implementation of an X Window server for PC based UNIX systems. It supports most PC video chipsets.

4.3 The VuSystem

The VuSystem [17] is used as the multimedia programming system. The VuSystem has been developed by the Telemedia, Networks and Systems Group at the MIT Laboratory for Computer Science. See [18] for a detailed description and reference to the VuSystem.

The VuSystem uses a media-flow architecture to process data. A media stream is processed by connecting different data processing modules, forming a directed media processing pipeline. Media data is held in dynamically-typed time-stamped units called payloads. The passing of payloads through the processing modules defines the flow of media data. The VuSystem also provides an interactive programming environment for real-time manipulation of media data [27]. A distributed programming extension to the VuSystem (VuDP) [22] was developed by Brent Philips to facilitate distributed media computation in which different modules can reside on different hosts on a network.

Figure 4-2 shows the structure of VuSystem applications. A VuSystem application is composed an in-band processing partition and an out-of-band processing partition.
4.3.1 In-Band Processing

The in-band processing partition handles the direct manipulation of live media data. It is arranged into processing modules that perform specific functions on media payloads. A module logically passes media payloads through its input ports and output ports. Processing modules are assembled in pipelines which define the flow of media payloads. The module data protocol defines the mechanism to transfer payloads ownership between modules.

There are three different kinds of modules:

- **Source modules** have one output port but no input port. They produce new media payloads for processing. Usually, they interface to media input devices like video capture cards.

- **Sink modules** have one input port but no output port. They are the end of the process pipeline. All media payloads are discarded after they are processed. Usually, they interface to media output devices like video displays.

- **Filter modules** have one or many input ports and output ports. They perform specific media processing functions on the payload. For example, a JPEG compression module compresses video frame payloads. A motion detector module signals the application program when motion is detected in a sequence of video frame payloads.

The in-band partition code is written in C++, for efficiency, while the out-of-band partition in Tcl, for flexibility.
Module Data Protocol

The module data protocol is the mechanism to pass payloads from an *upstream* module to a *downstream* module. It uses a ready/not ready protocol to communicate timing constraints between modules. Figure 4-3 shows the basic protocol to transfer a payload from an upstream module to a downstream module.

The upstream module calls the `Send` member function on its output port to send a payload. `Send` then calls the `Receive` member function on the downstream module. If the downstream module accepts the payload, it returns `True` to `Send`. `Send` then returns `True` to the upstream module to acknowledge the successful transfer of the payload. If the downstream module is not ready for the payload, it returns `False` to `Send` so that the upstream module would not try to send again. When the downstream module is ready for a new payload, it calls the `Idle` member function on its input port, which calls the `Idle` member function on the upstream module. `Idle` would send the payload to the downstream module.

Scheduler Interface

VuSystem modules invoke a number of services to schedule their `Work`, `Input`, `Output` and `TimeOut` class member functions.

- **Work** is called to perform long computations. The module calls the `StartWork` and `StopWork` scheduler interface functions to start or stop the calling of the `Work` member function.

- **Input and Output** are called to perform file input and output. After the module calls the `StartInput` scheduler interface function, the scheduler will call `Input` when the
specified file is ready for input. The module calls StopInput to stop the scheduled file input operation. Similarly, the module calls StartOutput to schedule Output to be called by the scheduler when the file is ready for reading and StopOutput to stop the scheduling.

- TimeOut is called to perform time-sensitive operations. The StartTimeOut scheduler interface function is used to schedule TimeOut to be called after a specified time has passed. StopTimeOut is used to cancel the scheduled event.

4.3.2 Out-Of-Band Processing

The out-of-band processing partition creates and controls the network of in-band processing modules and the graphical user-interface of the application. Since most of the functions are event-driven, performance is less important. The out-of-band code is programmed in Object Tcl [27], an object-oriented programming extension to Tcl [21]. Tcl is used because it has a simple interface to C and is easily programmable and extensible.

A Tcl object command is defined for each in-band processing module created in a VuSystem application. The out-of-band Tcl code controls the in-band module by using the Tcl object command for that module. In-band modules, on the other hand, can signal the out-of-band code of some in-band events through Tcl callbacks.

Example

Figure 4-5 shows a Tcl script fragment that creates a edge-detector example application. This application gets the video input from the CardCam video capture card on the mobile station. The video frames are processed by the edge-detector filter module and then displayed on the screen by the window sink module. This example illustrates how to create and connect in-band processing modules to form a media processing pipeline.

The script first creates an instance of the VsCardCamVideoSource module named m_source. The -scale parameter tells the module what video capture resolution to use. Then it creates an instance of the VvEdge filter module named m_edge. The -input parameter connects its input port to the output port of m_source. m_edge get video frames from its input port, perform an edge detection image processing function on them, and output the processed frames on its output port. Next, the script creates a windows sink
module named `m.sink`. The `-widget` parameter specifies the widget named `w.screen` to be the window to display the video frames on the screen. It also connects the input port of `m.sink` to the output port of `m.edge` using the `-input` parameter. Finally, the script starts all the modules by issuing the `start` command to the root of the modules hierarchy `w`, which in turn issues start to all of its children modules.

### 4.4 Media Synchronization

A key issue in multimedia systems is how to exactly control the media capture speed and playback speed. Audio and video data captured simultaneously are also played back simultaneously. Since media data are packeted up in payloads, each payload is individually played back at the appropriate time. VuSystem uses a system of media payload timestamps to synchronize media.

In VuSystem, every payload has a `StartingTime` payload descriptor member, which
carries the time of day at which the media data is captured. This serves as a permanent timestamp of the payload. It is stored to and retrieved from permanent storage with the payload. When a stream of payloads are ready to be played back, it goes through a VsReTime filter module. The VsReTime filter shifts the starting time of a sequence of payloads to the desired time of playback by adding a fixed time offset to the StartingTime payload descriptor. The time offset is calculated by taking the time difference between the time of capture and the current time plus a small buffer of the first payload of the sequence. This time offset is used to shift the StartingTime for the rest of the sequence of payload so that the sequence of payloads is played back at the rate at which they are captured. The media sink module then play back a payload at the time specified in the StartingTime descriptor. VsReTime can also make the media play back at a different rate than the capture rate by scaling the time offset according to the time at which the payloads are captured in the payload sequence.

4.5 Modules as Interfaces to Devices

The VuSystem uses different modules to separate different media processing functions. The modules' input and output ports pass payloads between modules. Modules also act as interfaces to the outside devices: the VuSystem source modules generate payloads from data produced by media capture devices; the VuSystem sink modules convert payloads into raw media data and output to media rendering devices.

The media input/output and network subsystem used in the mobile system use the module interfaces to connect to the VuSystem in-band processing pipeline. This provides a convenient interface for future expansion when new kind of I/O device become available.

4.6 Performance

To verify that the VuSystem module interface has enough bandwidth to handle the media flow of most media input/output devices, I measured the throughput of payloads passing from artificial video sources to artificial video sinks.

The source module used is the VsTestVideoSource module that very cheaply creates video frames of different sizes for testing. The sink modules used are the VsNullSink module, which simply destroy all the payloads it receives, and the VsWindowSink module, which
Table 4.1: VuSystem module interface frame rate with different artificial sources and sinks modules.

<table>
<thead>
<tr>
<th></th>
<th>VsTestVideoSource to VsWindowSink (frames/second)</th>
<th>VsTestVideoSource to VsNullSink (frames/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>320x240</td>
<td></td>
</tr>
<tr>
<td>gray</td>
<td>51.0</td>
<td>2740</td>
</tr>
<tr>
<td>color</td>
<td>51.0</td>
<td>2740</td>
</tr>
<tr>
<td>160x120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gray</td>
<td>51.0</td>
<td>2740</td>
</tr>
<tr>
<td>color</td>
<td>51.0</td>
<td>2740</td>
</tr>
</tbody>
</table>

Figure 4.1 shows the frame rates of video payloads passing from VsTestVideoSource to VsWindowSink, and from VsTestVideoSource to VsNullSink under two frame sizes. From the fast frame rate of the VsTestVideoSource to VsNullSink test, we can see that the time spent on passing payload between the modules is very low. Therefore, the VuSystem module interface has enough bandwidth to support very high speed devices.

Note the equal measured frame rates between different frame sizes. In the VsTestVideoSource to VsNullSink test, the passing of payloads between modules is done by passing a memory reference to a block of shared memory, not by copying memory. Therefore, the throughput does not depend on the frame size. In the VsTestVideoSource to VsWindowSink test, the video frames are actually copied to the X Server. The equal frame rates between different frame sizes suggest that the time spent is dominated by context switches between the X server and the VuSystem application.
Chapter 5

Video Input Subsystem

This chapter describes the video input subsystem, which provides a video source for the VuSystem running on the mobile station. The video input subsystem consists of a Quadrant CardCam PCMCIA video capture card [11], a CardCam device driver and a VuSystem CardCam video source module. The CardCam video source module accesses the video capture card through the device driver. It creates VuSystem video payloads from the captured video frames and feeds them to the rest of the VuSystem. (Figure 5-1)

A description of the video capture hardware is given in the next section. The design of the software architecture is then presented in Section 5.2. The implementation of the CardCam device driver and the VuSystem CardCam video input module are described in Sections 5.3 and Section 5.4 respectively. Performance results are discussed in the last section.

5.1 Video Capture Hardware

The mobile nature of the system presents some constraints on the choice of video capture hardware. The video hardware needs to be small and light to be used with the mobile station. On the other hand, it should provide enough performance for use with mobile VuSystem applications.

The Quadrant CardCam PCMCIA video capture card was chosen as the video capture device. Some laptop computers\(^1\) have a build-in video capture device. However, I use a PCMCIA video capture card instead because it is highly portable and can be used with any

---

\(^1\)For example, the IBM ThinkPad 755CD and 760CD.
laptop computer with a PCMCIA slot. (See Section 2.6 for a description of the PCMCIA standard.) It lets users choose any laptop computers according to their needs and still be able to use our video capture hardware and software. The video capture hardware can also be upgraded independently as technology advances. The CardCam video capture card connects to any composite or S-video (Y/C separated) source for video input and accepts both NTSC and PAL video signal. It has hardware hue, saturation, brightness and contrast control. It uses a Brooktree Bt812[4] video decoder, which provides acceptable performance.

The video capture card is connected to a miniature camera attached to the top of the mobile station for video input signal. The camera chosen is a miniature black and write camera that measures 1.15"x1.15"x0.5" and weights 0.8 ounce without battery. (Figure 5-2) While the camera is black and write, the image quality is excellent and is adequate for most applications. Smaller and lighter color cameras may be available in the future.

There are definitely tradeoffs between portability and performance in the selection of video capture hardware. As technology advances, new video equipment will allow a wider

---

3The camera outputs 380 lines of resolution. It uses a high-density 1/4" micro CCD chip and has light sensitivity of 1/2 lux.
range of selections for different uses.

5.2 Software Architecture

Several software layers lie on top of the PCMCIA adaptor. The PCMCIA Card Services for Linux, written by David Hinds, [10] provides a general interface for PCMCIA device drivers to access and control the PCMCIA host adaptor and devices. It also provides facilities for managing standard PCMCIA features like hot-swapping, dynamic resources allocation and advanced power management. (See Section 2.6.1 for a description of the Linux PCMCIA Card Services software.) On top of the Linux PCMCIA Card Services, the CardCam device driver provides a general interface for user-level programs to access the video capture card. The VuSystem video source module captures video data through the CardCam device driver and produces VuSystem video payloads for processing by other VuSystem modules. (Figure 5-1)

5.3 CardCam Device Driver

The CardCam device driver, developed as part of this work, provides a common interface for user-level programs (like the VuSystem) to access and control the video capture card. The device driver provides facilities to capture video and to control the input port, input video standard, capture size, brightness, contrast, saturation and hue of the video capture device. This PCMCIA device driver differs from a regular device driver in that it also supports most PCMCIA features like dynamic allocation of memory, I/O addresses and IRQ lines.
<table>
<thead>
<tr>
<th>ioctl command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>FG_SIZE_SET</td>
<td>Set the video capture size.</td>
</tr>
<tr>
<td>FG_SIZE_GET</td>
<td>Get the video capture size.</td>
</tr>
<tr>
<td>FG_STD_SET</td>
<td>Set the input video signal standard.</td>
</tr>
<tr>
<td>FG_STD_GET</td>
<td>Get the input video signal standard.</td>
</tr>
<tr>
<td>FG_PORT_SET</td>
<td>Set the video input port.</td>
</tr>
<tr>
<td>FG_PORT_GET</td>
<td>Get the video input port.</td>
</tr>
<tr>
<td>FG_BRIGHT_SET</td>
<td>Set the hardware brightness level.</td>
</tr>
<tr>
<td>FG_BRIGHT_GET</td>
<td>Get the hardware brightness level.</td>
</tr>
<tr>
<td>FG_CONTRAST_SET</td>
<td>Set the hardware contrast level.</td>
</tr>
<tr>
<td>FG_CONTRAST_GET</td>
<td>Get the hardware contrast level.</td>
</tr>
<tr>
<td>FG_SAT_SET</td>
<td>Set the hardware saturation level.</td>
</tr>
<tr>
<td>FG_SAT_GET</td>
<td>Get the hardware saturation level.</td>
</tr>
<tr>
<td>FG_HUE_SET</td>
<td>Set the hardware hue level.</td>
</tr>
<tr>
<td>FG_HUE_GET</td>
<td>Get the hardware hue level.</td>
</tr>
<tr>
<td>FG_RESET</td>
<td>Hardware reset</td>
</tr>
</tbody>
</table>

Table 5.1: CardCam device driver I/O control commands.

5.3.1 Interface to Client Programs

The CardCam device driver uses a UNIX device special file as an interface to client programs. A device special file is an abstraction supported by the Linux kernel Virtual File System (VFS) to allow user programs to access a device driver like a regular file. When the CardCam device driver module is loaded upon card insertion, the device special file /dev/cardcam is created. User programs access the device using normal file operations on the device special file. For example, client programs use open to setup CardCam for video capture, close to shutdown the device, read to read in each captured video frame, and ioctl (I/O control) to issue control commands to the video capture hardware. See Table 5.1 for a list of available ioctl commands.

Example

Figure 5-3 shows a code fragment that captures a 320 pixels by 240 pixels 8-bit color video image:

Here, the program first opens the device file /dev/cardcam as a read-only file and assigns the returned file descriptor to variable fd. fd is used as an identifier for the opened device file in subsequent read, ioctl and close calls. The program then uses the FG_SIZE_SET ioctl command to set the video capture size to be 320x240 pixels (size 2) and the FG_OUTPUT_SET
int fd;
char image[320][240];

if (fd=open("/dev/cardcam", 0_RDONLY) < 0) {
    perror("Could not open frame grabber, giving up");
    exit(-1);
}
ioctl(fd, FG_SIZE_SET, 2);
ioctl(fd, FG_OUTPUT_SET, OUTPUT_COLOR);
if (read(fd, &image, sizeof(image)) < 0) {
    perror("read error");
    exit(-1);
}
close(fd);

Figure 5-3: Sample code to capture an image from the CardCam device driver.

The ioctl command to set the output video encoding to be 8-bit color (OUTPUT_COLOR). Then it reads the captured image from the device to a buffer image by the read command. Finally, it closes the device file using the close command.

5.3.2 Internal Operations

The internal operations of the device driver can be best illustrated by describing what happens when different device driver events occur. There are six major events: card insertion, card release, device open, device read, device I/O control and device close.

Card insertion and card release are PCMCIA card events triggered usually by the physical insertion and removal of the card. These card events notify the device driver so that it can handle the changes in the state of the PCMCIA card. Device Open, read, I/O control and close events, however, are file access and control events generated by the client program when accessing the device special file.

Each event is described in details in the following sections.

5.3.3 Card insertion

When the CardCam video capture card is inserted into the PCMCIA slot, the PCMCIA card demon (cardmgr) reads the Card Information Structure (CIS) on the card and identifies

---

5 These card events can also be generated by software, although it is less common.
4 The Card Information Structure describes the card and how it should be configured. PCMCIA card manufacturers choose the information to be included in CIS. For example, it can contain information about the manufacturer, type of card, card memory addresses, I/O port used, power requirements, timing require-
what type of card it is by reading the manufacture and product identification strings. Figure 5-4 shows the portion of PCMCIA configuration file /etc/pcmcia/config that describes the CardCam video capture card. According to the PCMCIA configuration file, it loads the CardCam device driver module cardcam_cs.o and executes a script /etc/pcmcia/fgrabber to start or stop the CardCam device. The fgrabber script creates the device special file /dev/cardcam.

device "cardcam_cs"
module "cardcam_cs"
start "/etc/pcmcia/fgrabber start %d%%m% %n%%" 
stop "/etc/pcmcia/fgrabber stop %d%"

card "Quadrant CardCam-Video In"
version "Quadrant International", "CardCam-Video In", "Ver 2.00", ""
bind "cardcam_cs"

Figure 5-4: CardCam entries in the PCMCIA configuration file.

When the device driver module cardcam_cs.o is loaded, the init_module [init_cardcam] function is called, which registers the driver with Driver Services by calling register_pcmcia_driver. This informs Driver Services that a client driver is available to be bound to a socket. cardcam_attach and cardcam_detach functions are passed in as entry points to create and destroy an instance of the device driver. 

The device driver also performs a register_chrdev to register a character device with the kernel Virtual File System (VFS). A file_operation structure is passed in register_chrdev as an argument. The file_operation structure links various file operations to the corresponding handler functions so that VFS knows how to route file operations on the device file to the appropriate functions in the device driver. Figure 5-5 shows the content of the file_operation structure in the device driver.

This structure tells the kernel VFS to call fg_read upon a read request, fg_ioctl upon an ioctl request, fg_open upon an open request, and fg_close upon a close request. It returns an EINVAL error upon any other file requests.

After init_module registers the device with Driver Services, Driver Services calls cardcam_attach entry point to create an instance of the driver. It allocates memory spaces

ments, etc.

5In theory, you can have multiple instances of the device driver controlling multiple CardCam devices at the same time. However, the CardCam device driver currently does not support multiple devices yet.
static struct file_operations fg_fops = {
    NULL,    /* lseek */
    fg_read, /* read */
    NULL,    /* write */
    NULL,    /* readdir */
    NULL,    /* select */
    fg_ioctl, /* ioctl */
    NULL,    /* mmap */
    fg_open, /* open */
    fg_close, /* close */
    NULL     /* fsync */
};

Figure 5-5: The file_operation structure in CardCam device driver.

and initializes various data structures for managing one device:

1. dev_link_t structure holds information about the I/O port mapping between the
host and the PCMCIA device, interrupts setup, and general socket configuration (like
IRQ enable, Vcc setting, memory and I/O interface information) cardcam_attach
initializes the structure but does not actually allocate the resources. The actual
allocation is done in cardcam_config when the card insertion event is received.

2. local_info_t structure contains local data for cardcam. It contains the current I/O
port and IRQ setting, image width, height, size, port, video signal standard, output,
etc and is initialized to default values.

3. client_reg_t structure is used by Card Services to link the driver to a PCMCIA
socket.

cardcam_attach registers with Card Services and connects the client with an appropri-
ate socket. One piece of information it supplies is an event handler cardcam_event and
an event mask of interested card events. When the card status changes, Card Services
will notify client by calling the event handler cardcam_event if it is one of the interested
events. The cardcam_event function is a large case statement that invokes different actions
upon different card events. The events that are interesting to the CardCam driver is card
insertion, card removal, card reset, card suspend and card resume.

Since a PCMCIA card has been inserted, Card Services calls cardcam_event with the
CS_EVENT_CARD_INSERTION event, which then calls cardcam_config. cardcam_config con-
figures the PCMCIA socket, and does the actual resources mapping between the PCMCIA
card and the host system so that the device is available to the system. It calls Card Services to request IO ports, an IRQ interrupt line, and a memory window. [I/O: It will try from 0x300 to 0x400 and allocate the first available 16 IO ports. Memory: maps the card memory to the host memory space. (size=0x1000).] It also initiates the color conversion lookup table for YCrCb to 8-bit pseudo color conversion.

5.3.4 Card release

Upon CS_EVENT_CARD_REMOVAL event, cardcam_event is called by Card Services to start the card removal process. The process is somewhat parallel to card insertion. First, cardcam_release is called to de-register the device and release the PCMCIA configuration. It first checks if the device is still currently opened. If so, it delays the release until it is closed. Then it calls Card Services to release all the host resources (memory space, I/O ports, IRQ lines) used by the device driver. cardcam_detach is called by Device Services to delete the driver instance, de-register with Card Services, and free the memory used to manage the PCMCIA device. The CardCam device driver module is then removed. cleanup_modules is called to de-register the character device cardcam from the kernel VFS using unregister_chrdev. Finally, the fgrabber script removes the device special file /dev/cardcam.

5.3.5 Device open

The fg_open function is registered with the kernel VFS to be called whenever the device file is opened. fg_open performs a hardware reset of the card and then initializes all the hardware register with default values so that it is ready to capture images.

5.3.6 Device read

The function fg_read is called when the client tries to read a captured video frame from the device special file. It first resets the field memory by writing to the board control register. Then, it enables the vertical blank interrupt, which generates an interrupt on the beginning of every new field of video output (60 fields per second on NTSC signal) and puts itself to sleep (by calling interruptable_sleep_on). When there is a vertical blank signal, the interrupt will cause the interrupt handler cardcam_interrupt to be invoked. The handler resets the interrupt and wakes up the sleeping process. This sleep-wake up
process is repeated until it encounters an even field to ensure the video capture card has
enough time to capture the entire image. The captured image is read from the memory-
mapped FIFO and converted to the requested color mode. The resulting image data is
copied to the buffer in user-space. The read function returns the byte count of the captured
image.

Color Conversion

The CardCam video capture card captures images in 16-bit \( YCrCb \) color format.\(^6\) The
\( YCrCb \) data is then converted to 24-bit \( RGB \), 24-bit \( BGR \), 8-bit pseudo-color or 8-bit
grayscale for use in VuSystem.

Each pixel is represented in 4:2:2 \( YCrCb \) format. Pairs of pixels are encoded in a 32-bit
value \([Y_1, Cr, Y_2, Cb]\) in which \( Y_1 \), \( Y_2 \), \( Cr \) and \( Cb \) are 8-bit values. The \( YCrCb \) value of the
first pixel is \([Y_1, Cr, Cb]\) and the second pixel is \([Y_2, Cr, Cb]\).

Grayscale output is just output the \( Y \)-values of the image. For 8-bit color output, the
driver uses table lookup on the most significant 5-bits of the \( Y \), \( Cr \) and \( Cb \) values of each
pixel. For 24-bit color outputs, \( YCrCb \) to \( RGB \) conversion is calculated by the following
formula:

\[
\begin{bmatrix}
R \\
G \\
B
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 1.402 \\
1 & -0.34414 & -0.71414 \\
1 & 1.772 & 0
\end{bmatrix}
\begin{bmatrix}
Y \\
Cb \\
Cr
\end{bmatrix}
\]

To speed up the calculation, the driver uses integer multiplication and division to ap-
proximate the answer.

5.3.7 Device I/O control

\texttt{fg_ioctl} is called to handle device file \texttt{ioctl} calls. The \texttt{ioctl} command is used to control
the device special file when normal file read and write are not adequate. Table 5.1 lists the
cardcam ioctl commands and a brief description of their functions. See appendix B cardcam
device reference for a detailed description of each command.

\(^6\)The \textit{Brooktree Bt812} video decoder used in CardCam can output 24-bit RGB color. However, the
CardCam capture card only connects the first 16 bits of the 24-bit output from Bt812 to the PCMCIA bus,
so it's impossible to get the entire 24-bit RGB output.
The `fg_ioctl` function consists of a large case statement which carries out different functions upon different ioctl commands. Most of the ioctl commands changes or queries the Bt812 control registers settings, or changes the `local_dev_t` structure that holds local information of the device.

### 5.3.8 Device close

`fg_close` is called when the device file is closed. It is an empty function since no extra operation is needed when closing the device.

### 5.4 CardCam Video Source Module

The VuSystem CardCam video source module creates video frame (VsVideoFrame) payloads from the captured images and feeds them to other VuSystem modules. It uses the CardCam device driver to capture the images and to control the hardware device. It provides a control panel for users to interactively control the video capture device.

#### 5.4.1 Module Data Protocol

The CardCam video source module is the start of the in-band media processing. It creates VuSystem video frame payloads and pass them to downstream modules for processing.

46
The passing of payload from upstream modules to downstream modules is governed by the *module data protocol*. It defines a ready/not ready protocol for the passing of payloads to other modules.

The in-band media processing of the CardCam video source module is similar to other video source modules\(^7\) and the X Window sink module *VWindowSink*. All of these modules need to schedule time-sensitive operation with each payload. For a video source module, it captures frames at specific intervals. For a window sink module, it displays images at specific rates.

The video capture operation is performed in the *Timeout* C++ class member function. The module schedules time-sensitive operation with the VuSystem scheduler through *StartTimeOut*. After the specified time has passed, the VuSystem scheduler calls the *Timeout* C++ member function to perform the scheduled operations.

Figure 5-6 shows a diagram of *VsCardCamVideoSource* module indicating the interactions among the C++ class member functions in the module, the VuSystem Scheduler and the Linux kernel:

*Timeout* first checks if the module still holds any unsent payload. If it is not holding any payload, it calls either *Depth8Color*, *Depth24Color*, or *Depth8Gray* depending on the current color depth and color mode. These functions allocate a memory block to hold one frame, read a video frame from the CardCam device special file to the memory block, and to create a new video frame payload of the appropriate type. Finally, *Timeout* calls *Idle* to send the payload to the output port.

*Idle* first checks if the module is holding any unsent payload. If there is an unsent payload, it calculates the next time-step, updates the duration of the payload, and calls *Send* on the output port to send the payload to the downstream module. If the payload is sent successfully, it clears the payload from the module and calls *StartTimeOut* to schedule the next capture operation at the next time-step.

When the time specified in *StartTimeOut* arrives, the VuSystem scheduler calls *Timeout* and starts the video capture cycle again.

---

\(^7\)Currently there are VuSystem video source modules for Sun VideoPix, SunVideo and Vidboard video capture devices.
Start and Stop

The Start C++ class member function is called to perform module initialization at the start of in-band media processing. The Stop C++ class member function is called at the end of in-band media processing.

Start first opens the CardCam device file if it is not already opened. It then initializes the scale, port, video standard, brightness, contrast, saturation, hue, output, and encoding values of the CardCam device through a series of ioctl calls. Next, it calls VsEntity::Start to invoke the Start C++ class member functions of any children of this module. Next, Start resets the in-band processing by calling StopTimeOut to cancel any scheduled timeout operation and remove any payloads that it is holding. Start then creates a VsStart payload with the current time as the starting time, and calls Send on the output port to send it downstream. This payload indicates the start of a payload sequence. Finally, it starts the in-band data flow by calling Idle if the mode parameter is false.

Stop first calls VsEntity::Stop to cause the Stop C++ class member functions of any children of this module to be called. It then cancels any scheduled timeout operation with StopTimeOut, deletes any payload the module is still holding, and closes the CardCam device file. If mode parameter is false, indicating a VsFinish payload should be sent, Stop creates a new VsFinish payload with the current time as the starting time and sends it downstream.

5.4.2 Module Object Subcommand

VuSystem applications can manipulate the VsCardCamVideoSource module from the Tcl application script through module object subcommands. They are implemented as Tcl command procedures. Table 5.2 lists the object subcommands for VsCardCamVideoSource module and their functions. See Appendix A.1 for a reference manual of the VsCard-CamVideoSource module.

5.4.3 Subcommand Definition

Most of the subcommands are just wrappers for the corresponding cardcam device ioctl commands that let the VuSystem Tcl application script to control the video capture device.

Subcommands are Tcl command procedures and are declared C++ friend procedure to
<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathname</td>
<td>Get and set the device file pathname.</td>
</tr>
<tr>
<td>port</td>
<td>Get and set the video input port.</td>
</tr>
<tr>
<td>std</td>
<td>Toggle between NTSC and PAL video signal standard.</td>
</tr>
<tr>
<td>color</td>
<td>Toggle between color and grayscale.</td>
</tr>
<tr>
<td>brightness</td>
<td>Get and set the brightness level.</td>
</tr>
<tr>
<td>contrast</td>
<td>Get and set the contrast level.</td>
</tr>
<tr>
<td>saturation</td>
<td>Get and set the saturation level.</td>
</tr>
<tr>
<td>hue</td>
<td>Get and set the hue level.</td>
</tr>
<tr>
<td>scale</td>
<td>Get and set the video capture size.</td>
</tr>
<tr>
<td>depth</td>
<td>Toggle between 8 and 24 bit color depth.</td>
</tr>
<tr>
<td>frameRate</td>
<td>Get and set the video capture frame rate.</td>
</tr>
<tr>
<td>byteOrder</td>
<td>Toggle between MSB first and LSB first byte order.</td>
</tr>
<tr>
<td>encoding</td>
<td>Get and set the video signal encoding scheme.</td>
</tr>
</tbody>
</table>

Table 5.2: VsCardCamVideoSource module object subcommand.

the module class so that they can manipulate the private member function of the module. Figure 5-7 lists the code of the brightness subcommand as an example of how a typical subcommand is defined.

The VsCardCamVideoSourceBrightnessCmd function defines the brightness Tcl subcommand. First, the program casts the client data parameter cd to a pointer to the module. Next, it checks if there are more then one parameters and reports the error through VsTclErrArgCnt. If there is only one parameter, which is the brightness level, the function converts the parameter to an integer by VsGetInt. The program then scales the brightness to a -63 to 64 scale and sets the brightness instance variable to the new brightness level. Then it uses the ioctl function to change the brightness setting on the video capture hardware. If there is any error, it returns TCL_ERROR and the error message. If there is no parameter, the function will just return the current brightness.

In order to have the object subcommand to be linked into the application shell, the program calls CreateOptionCommand in the C++ constructor for the module. It links the function VsCardCamVideoSourceBrightnessCmd to the Tcl object subcommand brightness for the VsCardCamVideoSource module.

Control Panel

Figure 5-8 shows the control panel for the VsCardCamVideoSource module. It provides a visual interface that allows users to interactively adjust the option subcommands of the
int VscardCamVideoSourceBrightnessCmd(ClientData cd, Tcl_Interp* in, int argc, char* argv[])
{
    VscardCamVideoSource* src = (VscardCamVideoSource*)cd;
    if (argc > 2) return VstclErrArgCnt(in, argv[0], "?percent?" );
    if (argc > 1) {
        int percent;
        if (VstclGetInt(in, argv[1], &percent) != TCL_OK) return TCL_ERROR;
        if (percent < -50 || percent > 50) return VstclErrBadVal(in, "-50 - 50 percent", argv[1]);
        src->brightness = (percent*128)/100;
        if (src->fd >= 0 && ioctl(src->fd, FG_BRIGHT_SET, src->brightness) < 0) {
            Tcl_AppendResult(in, src->name(), ",: ioctl FG_BRIGHT_SET ",
                             src->pathname, ",: ", strerror(errno), 0);
            return TCL_ERROR;
        }
    }
    return VstclReturnInt(in, (src->brightness*100)/128);
}

VscardCamVideoSource::
VscardCamVideoSource(Tcl_Interp* in, VsEntity* pr, const char* nm)
{
    CreateOptionCommand("brightness", VscardCamVideoSourceBrightnessCmd,
                        (ClientData) this, 0);

    // ... code to register other subcommands
}

Figure 5-7: The source code for the brightness subcommand in the VscardCamVideoSource module.

module. Every module that supports a control panel defines a panel Tcl class procedure which constructs the graphical user interface.

Figure 5-9 shows a simplified version of the definition of the panel Tcl class procedure for the VscardCamVideoSource module. It constructs the graphical user interface for the brightness option subcommand defined above.

The panel Tcl class procedure first creates a Form widget as the main widget. Then it creates a Label widget with the label "CardCam" and no border. Next, the VslabeledScrollbar creates a control panel with a scroll bar for adjusting a continuous value. These are the parameters that specify properties of the control panel: -label sets the label of the panel to be "Brightness"; -value sets the initial value of the panel to be the result of evaluating the brightness subcommand of the module; -continuous specifies the scroll bar to be adjustable continuously; -converter specifies
that `vsRoundingLinearConverter` to be used as the procedure to convert from scroll bar unit to module unit, an integer between -50 to 50; `-inverter` specifies the procedure `vsLinearInverter` to be used to convert from module unit back to scroll bar unit; `-callback` specifies that the `brightness` subcommand to be called when the user adjusts the scroll bar; and `-fromVert` sets the position of the panel to be vertically below the label widget. Control panels for the other option subcommands are constructed similarly.

5.5 Performance

This section reports performance results of the video input subsystem. As I will show, video capture is a bottleneck for a typical VuSystem application because of the relatively low performance of the current video capture hardware. A careful performance study would enable us to improve the overall system performance.

I made three measurements on the video capture subsystem:

1. *Video capture device throughput* measures the CardCam video capture card hardware and device driver performance.

2. *VuSystem video capture throughput* measures the throughput of a simple video capture application under various setups.
VsCardCamVideoSource instanceProc panel {w orient args} {
    Form $w
    Label $w.label 
        -label "CardCam" 
        -borderWidth 0

    VsLabeledScrollbar $w.brightness 
        -label "Brightness" 
        -value [$self brightness] 
        -continuous [true] 
        -converter "vsRoundingLinearConverter -50 50" 
        -inverter "vsLinearInverter -50 50" 
        -callback "[$self brightness]" 
        -fromVert $w.label

    ## definition of other control buttons.
}

Figure 5-9: The script fragment that construct the control panel for the brightness option command in VsCardCamVideoSource module.

3. **VuSystem video capture latency and jitter** measures the delay between video capturing time to video display time in a simple video capture application.

### 5.5.1 Video Capture Device Throughput

The throughput of the CardCam video capture card and the CardCam device driver was measured for various frame sizes and color modes.

#### Experimental Setup

A simple program was written to measure the time it took to capture 100 video frames by reading video frames directly from the device driver. The program only measured the time to read (capture) video frames. The device setup time (open and ioctl operations) was not included. I configured the device driver to capture 320x240 pixel and 160x120 pixel frames with five different output modes:

1. **null**: This is like a normal capture except that it does not actually read the captured frame data from the card into main memory. This gives the baseline for the fastest possible frame rate.

2. **raw**: This mode outputs the captured data without any processing. The raw image data is in 16-bit YCrCb format.
\begin{tabular}{|c|c|c|c|}
\hline
frame size & output mode & ThinkPad 760C frames/second & Digital DECpc frames/second \\
\hline
320x240 & null & 30 & 30 \\
& raw & 7.5 & 6 \\
& 8-bit gray & 7.5 & 6 \\
& 8-bit color & 7.5 & 5 \\
& 24-bit RGB & 6 & 3 \\
\hline
160x120 & null & 30 & 30 \\
& raw & 30 & 15 \\
& 8-bit gray & 30 & 15 \\
& 8-bit color & 25 & 15 \\
& 24-bit RGB & 15 & 10 \\
\hline
\end{tabular}

Table 5.3: CardCam device driver video capture frame rate

3. 8-bit gray: This mode converts the data to 8-bit grayscale.

4. 8-bit color: This mode converts the data to 8-bit color format.

5. 24-bit RGB: This mode converts the data to 24-bit RGB color format.

Measurements were performed on an IBM ThinkPad 760C (Intel 90 MHz Pentium laptop) and a Digital DECpc (Intel DX2 50 MHz workstation).

Results

Table 5.3 shows the measured frame rates under different configurations. Note that since the NTSC signal has 30 frames per second, the video capture frame rates are in integer fractions of 30. This would “round off” the frame rates to discrete jumps.

The results indicate that there is little performance difference between 8-bit gray and 8-bit color. This shows that the cost of $YCrCb$ to 8-bit color conversion is cheap because of the use of table lookup. On the other hand, the 24-bit $RGB$ performance is much lower because the conversion is done by integer arithmetic operations, which is more costly.

Also, notice that the video capture frame rate does not scale up with processor speed. The CPU performance of the ThinkPad 760C is more then two times faster than that of the DECpc. However, the video capture performance of the ThinkPad 760C is no more then two times faster than that of the DECpc. The reason is because most of the video capture time is spent in transferring data from the memory mapped FIFO to main memory. Since bus speed does not scale up with processor speed, the I/O intensive task does not scale up either.
Figure 5-10: Simple program to measure VuSystem video capture throughput

5.5.2 VuSystem video capture throughput

I measured the CardCam video capture performance as seen by the VuSystem.

Experimental Setup

Figure 5-10 shows the in-band modules of the test program to measure the relative time spent on capturing the video frame and displaying it on the screen. VsVideoSource module creates video payloads and feeds them to the VsRateMeter module, which then measures the frame rate of the video stream. The payloads are displayed at the VsVideoSink module. The VsVideoSource and the VsVideoSink modules are composite modules consisting of all video sources modules and all video sink modules respectively. In this experiment, we used the VsCardCamVideoSource and the VsTestVideoSource modules from VsVideoSource as the video source: VsCardCamVideoSource captures video frames using CardCam video capture card; VsTestVideoSource very cheaply generates video payload of different sizes. I used the VsWindowSink and the VsNullSink modules from VsVideoSink as our video sink: VsWindowSink displays the video frame on the X Window server; VsNullSink simply discards the video payload. I tested all four source-sink configurations to measure the time spent on different modules on the program: VsCardCamVideoSource to VsWindowSink measures the total system throughput from video capture to video display on the screen; VsCardCamVideoSource to VsNullSink excludes the effect of video displaying on the screen and measures how fast the CardCam module can capture frames; VsTestVideoSource to VsWindowSink measures how fast VuSystem can display images on the X Window server; VsTestVideoSource to VsNullSink is the base case to measure the overhead of the test source module and overhead of module passing protocol.

Results

Table 5.4 shows the measured results. As we can see, the VuSystem throughput of capturing from CardCam and displaying on screen is close to the throughput of the Card-
<table>
<thead>
<tr>
<th></th>
<th>CardCam Source to WindowSink (frames/second)</th>
<th>CardCam Source to NullSink (frames/second)</th>
<th>TestVideoSource to WindowSink (frames/second)</th>
<th>TestVideoSource to NullSink (frames/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>320x240</td>
<td>7.50</td>
<td>7.50</td>
<td>51.0</td>
<td>2740</td>
</tr>
<tr>
<td>gray</td>
<td>7.33</td>
<td>7.5</td>
<td>51.0</td>
<td>2740</td>
</tr>
<tr>
<td>color</td>
<td>29.33</td>
<td>30.0</td>
<td>51.0</td>
<td>2740</td>
</tr>
<tr>
<td>160x120</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gray</td>
<td>15.0</td>
<td>28.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: VuSystem video capture frame rate with different source and sink modules.

Cam hardware performance measured in the last section. In 320x240 pixel color mode, the program achieves 7.33 frames per second while the CardCam hardware performance is 7.5 frames per second. In 160x120 pixel color mode, the program achieves 15 frames per second while the CardCam hardware performance is 25 frames per second. The time spent on displaying the images on the screen is relatively small. This is evident by the close throughput between CardCam source to WindowSink and CardCam source to null sink. The overall overhead of module passing protocol and scheduling is known to be very small since throughput from the test source to the null sink throughput is very high. (2740 frames per second)

This result shows that:

1. CardCam video capture is the bottleneck in simple VuSystem applications.

2. The overhead of video display, payload passing, and scheduling is very small.

5.5.3 VuSystem video capture latency and jitter

Video latency and jitter affect the perceptual performance of real time video applications. This experiment measures the delay between video capture and video display in a simple VuSystem program.

Experimental Setup

I used the same program when I measured the video capture throughput in the last section with VsCardCamVideoSource as the video source and VsWindowSink as the video sink. I instrumented the code to measure the latency between the time when a video frame is captured and the time when the timeout function in VsWindowSink is called to display the
image on the screen. I measured the latency under half-size and quarter-size screen and under various frame rates. Measurements were done using the IBM ThinkPad 750C.

Results

Figure 5-11 and Figure 5-12 show plots of video frames latency over the course of a 66 seconds experiment. Each point on the plot represents a captured frame. The x-coordinate is the video capturing time and the y-coordinate is the latency of that video frame.

In 320x240 pixel mode, the shortest latency is about 132ms. In 160x120 pixel mode, the shortest latency is about 53 ms. As shown on the graphs, the latency is affected by the video frame rate. In general, a lower frame rate has a slightly better latency. The variation in latency may be caused by the fact that the Linux scheduler stops scheduling while the CardCam device driver is capturing a video frame (in the kernel). If the frame rate is high, the chance that the device driver blocks a scheduled task (like displaying a video frame on the display) when capturing video frames is higher. Therefore, the latency would be higher.

Also, the latency increase steadily within a 33ms band in most of the test cases. This is caused by the CardCam device driver waiting for the even field of NTSC video signal when capturing every frame. Even field occurs 30 times per second, or 33ms per cycle. Since the VuSystem scheduler, which calls Timeout C++ class member function to displays video frames, is not exactly in sync with the field rate, the latency increases steadily and waits at most 33ms extra time. This 33ms extra latency is very small compared to the video frame rate and total latency. Note that occasionally there are frames that go beyond the 33ms band. These are the frames that will cause significant jitter. As we can see in the graphs, only a small number of frames are not displayed on the screen on time. This shows that the video jitter is very small.
Figure 5-11: Video capture latency in 320x240 pixel mode.
Figure 5-12: Video capture latency in 160x120 pixel mode.
Chapter 6

Audio Input/Output Subsystem

The VuSystem has been extended to support PC audio hardware on the mobile station and other PC workstations that are running Linux.\(^1\) PC audio support presents some new challenges because of the large variety of PC audio hardware. For example, my laptop uses a Sound Blaster compatible audio chip\(^2\) while my desktop PC uses a Microsoft Sound System sound card. Different types of audio hardware have different capabilities that must be accommodated.

The new audio source/sink modules are designed to be flexible enough to use with different audio hardware configurations. By utilizing the universal API of the UNIX Sound System device driver, I constructed a flexible module control panel for users to configure the different PC hardware easily.

The overall design of the audio input/output subsystem is presented in the next section. The design and implementation of the UNIX Sound System device driver, the VuSystem Linux Audio Source and Sink modules, and the Linux AudioFile server are then discussed.

6.1 Design

The VuSystem accesses the audio hardware for audio capture and playback through the audio source and sink modules respectively: Audio source modules create \texttt{VsAudioFragment} payloads from raw audio data and pass them to downstream VuSystem filter modules.

\(^1\)The VuSystem also supports Sun Audio and Dec Audio.

\(^2\)The IBM ThinkPad 760C laptop uses an MWave DSP chip for audio capture/playback. We use its Sound Blaster emulation mode under Linux.
Audio sink modules take in **VsAudioFragment** payloads and play them back through the audio hardware. As shown in figure 6-1, two pairs of audio source/sink modules provide audio input/output for the VuSystem:

- **Linux Audio Source/Sink Modules:** The **VsLinuxAudioSource** and **VsLinuxAudioSink** modules directly communicate with the audio hardware device driver during audio capture and playback.

- **AudioFile Source/Sink Modules:** The **VsAudioFileSource** and **VsAudioFileSink** modules act as an AudioFile client and use the AudioFile API to communicate with the Linux AudioFile server for audio capture and playback. AudioFile is a machine-independent and network-transparent audio system. The AudioFile server may reside on a local or remote machine.

By default, the VuSystem uses AudioFile for audio capture and playback because the same AudioFile modules can be used across different platforms\(^3\). More importantly, it

\(^3\)The AudioFile server is available for Digital Ultrix, Digital Alpha AXP, Sun Sparc and SGI. AudioFile is also fairly portable to other platforms
provides network-transparent access to remote audio devices very much like the way X Window System accesses remote displays. However, the host needs to have an AudioFile server installed in order to use the AudioFile modules. The alternate way of accessing audio device under Linux is to use the VuSystem Linux audio source/sink modules. They access the audio hardware directly and provide more control over the audio device than the AudioFile modules. The different audio source/sink modules are usually created by the VsAudioSource and VsAudioSink modules. The `-audioSource :af` parameter to VsAudioSource and VsAudioSink specifies the use of AudioFile source and AudioFile sink modules respectively, whereas the `-audioSource: Linux` parameter specifies the use of Linux audio source and sink modules respectively.

Both the AudioFile server and the Linux audio modules access the audio hardware through the UNIX Sound System (USS) device driver that provides a universal audio and sound API for accessing different PC audio hardware configurations.

The design of the UNIX Sound System device driver and its API is described in the next section. The Linux Audio Source/Sink modules is described in section 6.3 and the Linux AudioFile server is presented in section 6.6.

### 6.2 UNIX Sound System

The UNIX Sound System\(^4\) [24] provides a universal audio and sound programming API for Linux and other operating systems\(^5\) It consists of a collection of sound device drivers for major PC sound hardware configurations.\(^6\) Client programs access the sound cards through a set of special device files. The major device files include:

- `/dev/mixer`: This is the mixer device on the card. It controls volumes of different channels. Not all sound cards have mixer devices.

- `/dev/dsp`: This is the main digitized voice device. Client programs can record and playback digitized voice by reading from and writing to this device file.

---

\(^4\)The UNIX Sound System was developed by Hannu Savolainen. It was formerly known as Linux Sound Driver, VoxWare Sound Driver and TASD.

\(^5\)Unix Sound System is originally developed for Linux. It is currently being ported to FreeBSD, SCO, UnixWare and AIX.

\(^6\)The major supported sound cards include SoundBlaster, ProAudio Spectrum, Advanced Gravis UltraSound and Roland MPU-401 MIDI card.
<table>
<thead>
<tr>
<th>Mixer channel</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOUND_MIXER_VOLUME</td>
<td>master output volume level.</td>
</tr>
<tr>
<td>SOUND_MIXER_TREBLE</td>
<td>treble level of all output channels.</td>
</tr>
<tr>
<td>SOUND_MIXER_BASS</td>
<td>bass level of all output channels.</td>
</tr>
<tr>
<td>SOUND_MIXER_SYNTH</td>
<td>volume level of the synthesizer input.</td>
</tr>
<tr>
<td>SOUND_MIXER_PCM</td>
<td>output level of the PCM digitized voice device.</td>
</tr>
<tr>
<td>SOUND_MIXER_SPEAKER</td>
<td>output volume for the PC speaker signals.</td>
</tr>
<tr>
<td>SOUND_MIXER_LINE</td>
<td>volume level for the line in jack.</td>
</tr>
<tr>
<td>SOUND_MIXER_MIC</td>
<td>volume level for the microphone input.</td>
</tr>
<tr>
<td>SOUND_MIXER_CD</td>
<td>volume level for the CD audio input.</td>
</tr>
<tr>
<td>SOUND_MIXER_IMIX</td>
<td>output volume of the recording monitor.</td>
</tr>
<tr>
<td>SOUND_MIXER_RECLEV</td>
<td>global recording level.</td>
</tr>
</tbody>
</table>

Table 6.1: Mixer channels.

- `/dev/audio`: This device is similar to `/dev/dsp` but uses μ-law encoding. It provides limited compatibility with the Sun `/dev/audio` device.

- `/dev/sequencer`: It provides access to the internal synthesizer devices of the sound cards. It can also access external music devices that are connected through the MIDI port of the sound card.

- `/dev/midi`: This device plays MIDI files.

VuSystem audio modules only use `/dev/mixer`, `/dev/dsp` and `/dev/audio` devices for audio capture and playback. Some programming information relevant to the development of the audio subsystem is described below. Please refer to [23, 24] for full USS programming references.

### 6.2.1 The Mixer Device

Sound card uses mixers to adjust volume levels for different inputs and outputs. However, not all soundcards have mixer devices. USS uses the device special file `/dev/mixer` to control the mixer. It will return ENXIO error upon ioctl calls if no mixer is present.

The mixer can query and adjust the volume level of different mixer channels. Each channel has a name and is assigned an integer from 0 to 30 in `soundcard.h`. Table 6.1 describes each channel recognized by the mixer. Note that even with the same channel, different soundcards may have a slightly different function of the channel.

To use the mixer device, the mixer program first opens the mixer device special file
<table>
<thead>
<tr>
<th>Mixer ioctl command</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixer query interface</td>
<td></td>
</tr>
<tr>
<td>SOUND_MIXER_READ_DEVDMASK</td>
<td>gets a bitmask of the available channels.</td>
</tr>
<tr>
<td>SOUND_MIXER_READ_RECDMASK</td>
<td>gets a bitmask of the available recording devices.</td>
</tr>
<tr>
<td>SOUND_MIXER_READ_STEREODEV</td>
<td>gets a bitmask of the available stereo devices.</td>
</tr>
<tr>
<td>Getting and setting volume</td>
<td></td>
</tr>
<tr>
<td>SOUND_MIXER_READ(chn)</td>
<td>gets the volume of channel chn.</td>
</tr>
<tr>
<td>SOUND_MIXER_WRITE(chn)</td>
<td>sets the volume of channel chn.</td>
</tr>
<tr>
<td>Selecting recording sources</td>
<td></td>
</tr>
<tr>
<td>SOUND_MIXER_READ_RECSRC</td>
<td>gets a bitmask of the active recording sources.</td>
</tr>
<tr>
<td>SOUND_MIXER_WRITE_RECSRC</td>
<td>change the active recording sources.</td>
</tr>
</tbody>
</table>

Table 6.2: Mixer device ioctl commands.

/dev/mixer. Then it uses ioctl commands with the mixer file descriptor, command name, and command arguments, as arguments to control the mixer. Table 6.2 summarizes the functions of different mixer ioctl commands.

Different soundcards have different mixer capabilities. The USS mixer device file provides an interface for querying the capabilities of the mixer in order for programs to work with different hardware setups.

```c
int mask=0, volume=50, mixerfd=-1;

if ((mixerfd = open("/dev/mixer", O_RDWR, 0)) == -1) {
    perror("/dev/mixer open failed");
    exit(-1);
}

if (ioctl(mixerfd, SOUND_MIXER_READ_DEVDMASK, &mask) != -1) {
    if (mask & (1 << SOUND_MIXER_PCM))
        ioctl(mixerfd, SOUND_MIXER_WRITE(SOUND_MIXER_PCM), &volume);
}
```

Figure 6-2: Code fragment to set the PCM volume of a mixer device.

Figure 6-2 shows a code fragment that opens a mixer device and sets the PCM output level to 50. The code first opens the mixer device /dev/mixer in read-write mode and assigns the returned file descriptor to variable mixerfd. If mixerfd is -1, the code reports an open error and exits. Next, it uses the SOUND_MIXER_READ_DEVDMASK ioctl command to query the available channels. The call returns a bitmask in the variable mask. If the returned status of the call is not -1, the mixer is available. The code checks if the SOUND_MIXER_PCM channel is available by testing if the corresponding bit in mask is set. If
the channel is available, the code uses the `SOUND_MIXER_WRITE` ioctl command to set the
`SOUND_MIXER_PCM` channel to volume.

### 6.2.2 The Digitized Voice Device

The `/dev/dsp` or the `/dev/audio` device special file are used to capture or playback digitized voice. The two device files are very similar except that `dev/dsp` uses 8-bit unsigned encoding while `/dev/audio` uses logarithmic µ-law encoding. To capture or playback audio, applications can simply open the appropriate device file, then `read` or `write` to the file.

```c
int audiofd, len;
char buf[1024];

if ((audiofd = open("/dev/audio", O_RDONLY, 0)) == -1) {
    perror("/dev/audio open failed");
    exit(-1);
}

if ((len = read(audiofd, buf, 1024)) == -1) {
    perror("audio read error");
    exit(-1);
}
```

Figure 6-3: Code fragment to capture audio.

Figure 6-3 shows a code fragment that captures 1024 bytes of audio data in µ-law format. The code first opens `/dev/audio` for µ-law audio format in read-only mode and assigns the returned file descriptor to `audiofd`. If `audiofd` is -1, the code reports a file open error and exits. Next, the code uses the `read` command to capture 1024 bytes of data to the buffer `buf`. `read` returns the number of bytes read and is assigned to variable `len`. If `buf` is -1, the code reports a read error and exits.

Applications can also control various parameters of `/dev/dsp` device by using `ioctl` calls. Table 6.3 lists the `ioctl` commands available.

### 6.3 The Linux Audio Source Module

The Linux audio source module (VsLinuxAudioSource) is the VuSystem interfaces to capture audio with the sound card. It uses the USS audio interface for accessing the audio devices. The module also provides a user interface for controlling the audio device.
<table>
<thead>
<tr>
<th>ioctl command</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNDCTL_DSP_RESET</td>
<td>reset the device.</td>
</tr>
<tr>
<td>SNDCTL_DSP_SETFMT</td>
<td>sets the audio format.</td>
</tr>
<tr>
<td>SNDCTL_DSP_GETFMT</td>
<td>gets the supported audio formats.</td>
</tr>
<tr>
<td>SNDCTL_DSP_STEREO</td>
<td>select stereo or mono mode.</td>
</tr>
<tr>
<td>SNDCTL_DSP_SPEED</td>
<td>select the sampling speed.</td>
</tr>
<tr>
<td>SNDCTL_DSP_SYNC</td>
<td>wait for all data written to the device to be played.</td>
</tr>
</tbody>
</table>

Table 6.3: Audio device ioctl commands.

![Diagram of the Linux audio source module]

Figure 6-4: The Linux audio source module

### 6.3.1 Module Data Protocol

The Linux audio source module is similar to a simple file source module because the device special file essentially operates like a regular file. Unlike the VsCardCamVideoSource module described in Section 5.4, which uses the Timeout function to capture frames at specific time intervals, the VsLinuxAudioSource uses Input to perform read operation from the device file. This module uses StartInput to schedule the file input operation with the VuSystem scheduler. When the specified file becomes available for reading, scheduler calls Input to perform the file input. When no more data is necessary, Input calls StopInput to stop the scheduler from calling Input. The interactions between C++ class member functions in the VsLinuxAudioSource module is shown in figure 6-4.

Input first reads a small audio fragment of size 1024 bytes from the device file, creates a new VsAudioFragment payload containing the data, and then sends the payload downstream to the next module by calling the Send function on the output port. Input will keep on
reading audio fragments, creating new audio payloads and sending the new payloads until the downstream module refuses to accept the payload, or that the audio device file does not have enough data available. Input then checks if the module is holding any unsent payload. If it is, Input stops scheduling file input operations with StopInput.

Idle checks if there is a payload to be sent. It calls the Send function on the output port if there is a payload to be sent. If the payload is accepted, Idle clears the local reference to the payload and calls StartInput to schedule Input to be called when input is available from the audio device file.

Start and Stop

Similar to the CardCam source module, the Start C++ class member function is called to perform module initialization at the start of in-band media processing. The Stop C++ class member function is called at the end of in-band media processing.

Start first open the audio device file if it is not already opened. Start then calls VsEntity::Start to invoke the Start C++ class member functions of any children of this module. Next, Start calls StopInput to abort any scheduled input operation. Then, it creates a VsStart payload with the current time as the starting time, and calls Send on the output port to send it downstream. This would indicate that it is the start of a payload sequence. Finally, it calls Idle to start an input operation if the mode parameter is false.

Stop first calls VsEntity::Stop to cause the Stop C++ class member functions of any children of this module to be called. Stop then cancels any scheduled input operation with StopInput, deletes any payload the module is still holding, and closes all the audio device file that is still opened. If the mode parameter is false, indicating a VsFinish payload should be sent, Stop creates a new VsFinish payload with the current time as the starting time, and send it downstream.

6.3.2 VsLinuxAudioSource object subcommand

The VsLinuxAudioSource object creation command is used to create a VsLinuxAudioSource module. A set of VsLinuxAudioSource object subcommand is defined to control the hardware settings of the device. The definition of the object subcommands is very similar to that of the VsCardCamVideoSource module in Section 5.4.3. Table 6.4 lists the available subcommands.
<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathname</td>
<td>Get and set the device file pathname.</td>
</tr>
<tr>
<td>port</td>
<td>Get and set the audio input port. (Mic, line, CD or synthesizer)</td>
</tr>
<tr>
<td>lineVol</td>
<td>Get and set the volume for the line in jack.</td>
</tr>
<tr>
<td>micVol</td>
<td>Get and set the volume for signal from the microphone in jack.</td>
</tr>
<tr>
<td>cdVol</td>
<td>Get and set the volume level from the CD audio input.</td>
</tr>
<tr>
<td>imixVol</td>
<td>Get and set the volume of the recording monitor on the PAS16 cards.</td>
</tr>
<tr>
<td>synthVol</td>
<td>Get and set the volume of the build-in synthesizer chip.</td>
</tr>
<tr>
<td>recLevel</td>
<td>Get and set the global recording level.</td>
</tr>
<tr>
<td>gain</td>
<td>Get the set the input gain.</td>
</tr>
</tbody>
</table>

Table 6.4: VsLinuxAudioSource module object subcommand.

6.4 The Linux Audio Sink Module

The Linux audio sink module is the VuSystem interface to playback audio with the sound card. It uses the USS audio interface to access the audio devices.

6.4.1 Module Data Protocol

![Diagram of the Linux audio sink module]

Figure 6-5: The Linux audio sink module

The Linux audio sink module (VsLinuxAudioSink) is similar to a simple file sink module because the device special file essentially operates like a regular file. It uses Output to perform write operation to the device file. The module uses StartOutput to schedule the file output operation with the VuSystem scheduler. When the specified file becomes available for writing, the scheduler calls Output to perform the file output. When all data
<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathname</td>
<td>Get and set the device file pathname.</td>
</tr>
<tr>
<td>volume</td>
<td>Get and set the master output volume.</td>
</tr>
<tr>
<td>bass</td>
<td>Get and set the bass level of all output channels.</td>
</tr>
<tr>
<td>treble</td>
<td>Get and set the treble level of all output channels.</td>
</tr>
<tr>
<td>pcm</td>
<td>Get and set the output level for the digitized voice channel.</td>
</tr>
<tr>
<td>speaker</td>
<td>Get and set the output volume for the PC speaker signal.</td>
</tr>
<tr>
<td>gain</td>
<td>Get and set the output gain.</td>
</tr>
</tbody>
</table>

Table 6.5: VsLinuxAudioSink module object subcommand.

are written to the file, Input calls StopInput to stop the scheduler from calling Output. It also uses StartTimeOut to schedule the playback time of the audio fragment. The interactions between C++ class member functions in the VsLinuxAudioSink module are shown in Figure 6-5 and described below.

Receive checks if the module is holding any audio payload. If it is, and if the playing time of the payload has passed, then Receive calls StartOutput to cause Output to be called when the audio device file is ready for output and returns True to accept the payload. If the playing time of the payload is in the future, Receive rejects the payload and calls StartTimeOut to schedule TimeOut to be called when the playing time of the payload is reached.

Output first calculates the fragment of audio data in the audio payload that should be played back. It then writes as much audio data as it can to the file. If all the data in the payload is written, the payload will be deleted. Output checks payload to see if it is null. If it is, Output stops scheduling file output operations with StopOutput and calls Idle on the input port to indicate that the sink is to receive more data.

TimeOut simply calls Idle on the output port to receive more data since the playing time of the next audio payload has been reached.

6.4.2 Linux Audio Sink module object subcommand

Table 6.5 lists the available object subcommands for the Linux audio sink module. These subcommands control the hardware settings of the audio device.
6.5 Control Panel for the Linux Audio Modules

Since different audio hardware supports different capabilities, the control panel for the audio source and sink modules dynamically determine what channels the device has at run time and only display the controls for the available channels. Each channel has a subcommand to control it. If the subcommand returns an error when called, it means that channel is not available. By querying echo channel subcommand, the control panel can determine what channels are available.

6.6 The AudioFile System

AudioFile is a platform-independent, network-transparent audio driver developed by Digital Equipment Corporation's Cambridge Research Center. I ported the AudioFile server to Linux with Unix Sound System mixer support.

The VuSystem AudioFile source and sink modules are used without modifications with the Linux AudioFile server. The modules are similar to the VsLinuxAudioSource and VsLinuxAudioSink modules, but provide only input and output gain controls.

6.6.1 AudioFile Protocol

AudioFile is a network audio system that manages audio connections similar to how the X Window System manages video connections. An AudioFile server runs on a host accepts connections from AudioFile clients on the same host or on remote hosts.

6.6.2 Device Independent Audio

The AudioFile server code is divided into device independent audio (DIA) and device dependent audio (DDA). The DIA provides facilities that are common to all audio devices, like audio connection management and the handling of the AudioFile API. The DDA supports device specific code that implements the actual audio playback or capture with the audio device. To support a new platform, only the DDA part is changed to support the new hardware. The Linux DDA code is modeled after the Sparc DDA code. The Linux DDA uses /dev/audio for u-law audio data and /dev/dsp for linear audio encoded data. It uses /dev/mixer to control the volume and sampling rate of the device.
6.7 Summary

The audio input/output subsystem allows the VuSystem to capture and playback audio on the mobile system. Two sets of audio source/sink modules can be used to access the audio device: The AudioFile source/sink modules use the AudioFile System for a network-transparent audio device; The Linux audio source/sink modules directly access the sound hardware on the mobile system. The Unix Sound System device driver provides a universal API for the Linux audio source/sink modules and the AudioFile server to access the PC audio hardware.

The use of the Unix Sound System device driver and the AudioFile System should not notably affect the overall system performance. The Unix Sound System device driver is the standard driver to access PC sound hardware. The AudioFile System is in use with different platforms without performance problems.
Chapter 7

Wireless Network Subsystem

The wireless network subsystem provides wireless network connection between the mobile station and workstations on a wired local area network backbone. VuSystem applications running on the mobile station can transmit media data and other information to and from other desktop media servers or mobile nodes. This extends the reach of multimedia applications to places where no wired network connection is available or where mobility is required. Network filter modules provide in-band media connections between modules on different hosts. This allows different media filter modules to be distributed on different hosts to utilize different processing and networking capabilities of the machines.

The wireless network subsystem consists of three major parts:

- **Hardware**: The wireless network hardware used is based on the AT&T WaveLAN wireless network system.

- **Device driver**: The WaveLAN device driver emulates an Ethernet network interface for use by Linux upper layer (TCP/IP) and socket protocol modules.

- **VuSystem network modules**: VuSystem network modules use the Linux socket interface to create network connections for in-band media to flow between two different machines.

The wireless network hardware, network configuration, network device driver, and the VuSystem network modules are described in more details in the following sections.
7.1 Network Hardware

The wireless network hardware used is based on the AT&T WaveLAN wireless network system\(^1\), which has the following attributes:

- **Proven technology:** WaveLAN units have been in use in our research lab for over a year. People have encountered few problems and are generally satisfied with them.

- **Linux device driver support:** Anthony Joseph (in the MIT Parallel and Distributed Operating Systems Group) developed a WaveLAN PCMCIA device driver for Linux.

- **Adequate performance:** The WaveLAN has a two megabit per/second data rate. Under normal TCP network connection, it has 150k-170k byte/second throughput. (See Section 7.6 for some performance results.)

The WaveLAN uses Direct Sequence Spread Spectrum (DSSS) Technology to distribute information and power across a frequency spectrum range approved by the FCC (ISM band) for unlicensed operation. We used the 915MHz version of WaveLAN Network Interface Card (NIC) with the WavePOINT bridge that connected the wireless network segment to a wired Ethernet backbone.

7.2 Network configuration

Figure 7-1 shows the network configuration used. The WavePOINT access point serves as a transparent Media Access Control (MAC) bridge between the Ethernet backbone and a WaveLAN wireless network segment. (The communication zone of a WavePOINT MAC bridge defines a WaveLAN wireless network segment.) Each mobile node configured with a WaveLAN NIC communicates with its closest WavePOINT access point within the diameter of the communication zone defined by the strength of the signal. WavePOINT and WaveLAN support roaming, that is, the WaveLAN can maintain network connection when traveling from one network segment to another. However, the current WaveLAN device driver for Linux does not support this feature.

\(^1\)To be exact, we used the Digital RoamAbout wireless network system, which is another version of the AT&T WaveLAN wireless network system.
7.3 Software Architecture

Figure 7-2 shows the software architecture of the wireless network subsystem. The software layered on top of the WaveLAN PCMCIA NIC are similar to that for the CardCam PCMCIA video capture card (Section 5.2) The PCMCIA Card Service for Linux, written by David Hinds [10], manages the PCMCIA adaptors on the mobile node and provide facilities for accessing and controlling them. (See Section 2.6.1 for a description of the Linux PCMCIA system.) The WaveLAN device driver is build on top of the Linux PCMCIA Card Services. It creates an Ethernet network interface for client programs to access the WaveLAN. VuSystem network modules (VsTcpClient, VsTcpServer, VsTcpListener) uses the WaveLAN device driver Ethernet interface to provide VuSystem applications access to the network.

7.4 WaveLAN device driver

The WaveLAN device driver has been developed by Anthony Joseph.² I patched the driver to be used with PCMCIA Card Services version 2.7.6. Anthony is currently working on support for roaming and support for the Linux 1.3.x kernel and PCMCIA Card Services version 2.8.x.

The basic structure of the device driver is the same as the CardCam device driver described in Section 5.3. Special care is taken to configure the device driver. Figure 7-3

²Driver source is available at http://www.pdos.lcs.mit.edu/adj/wavelan.html.
shows the portion of the PCMCIA configuration file that configures the WaveLAN PCMCIA adapter.

```
device "wavelan_cs"
  module "wavelan_cs"
  start "/etc/pcmcia/wavenet start %d%"
  stop " /etc/pcmcia/wavenet stop %d%"

card "NCR WaveLAN/PCMCIA Adapter"
  version "NCR", "WaveLAN/PCMCIA", ",*"
  bind "wavelan_cs"
```

Figure 7-3: WaveLAN entries in PCMCIA configuration file.

It instructs that the module `wavelan_cs.o` be loaded, and the script "/etc/pcmcia/wavenet start <device name>" be executed upon card insertion. Since the network is not available before the card is inserted, all network configuration commands that require a real network connection should be delayed and put in the `wavenet` script. For example, the routing table, network interface, and nameserver should be configured in the `wavenet` script.
7.5 VuSystem Network Modules

The VsTcpServer, VsTcpClient and VsTcpListener filter modules use TCP connections to create in-band communication channels between modules on different machines. The VsTcpClient module provides an interface to the client side of a TCP connection. The VsTcpServer module provides an interface to the server side of a TCP connection. They are classified as filter modules because they both have an input port and an output port. Payloads passed in one end of the connection come out of the other end of the connection. They are based on the VsByteStream module, which knows how to convert payloads from and to sequences of bytes. The VsTcpListener module provides the function similar to the listen system call. It listens to new connection requests on a TCP port. When a connection request from a VsTcpClient module is received, VsTcpListener will creates a VsTcpServer module and connects it to the VsTcpClient module. Refer to [18] appendix A for a description of these modules.

Example

The following example illustrate how to use VsTcpClient, VsTcpListener and VsTcpServer modules to setup a two-way TCP in-band communication channel between two machines.

```
VsTcpClient $m1.tcpclient \n   -host $host \n   -port $port \n   -input "bind $m1.source.output"

$m1.tcpclient.output connect $m1.sink.input
```

Figure 7-4: Sample VuSystem program script fragment to create the client side of a TCP connection.

Figure 7-4 shows the program fragment that creates the client end of the TCP connection. First a VsTcpClient module named $m1.tcpclient is created with the following parameters: the -host parameter specifies the server hostname; the -port parameter specifies the server TCP port number; the -input parameter connects the module’s input port to the media source module’s output port $m1.source.output. Then the program connects the VsTcpClient module’s output port $m1.tcpclient.output to the media sink module’s
input port $m1.source.output to complete the two-way connection to and from the TCP client module.

```tcl
$m2 proc listenerCallback {args} {
    set obj [keyarg -obj $args]
    if {$obj != ""} {
        $self.listener destroy
        $self.source.output connect $obj.input
        $obj.output connect $self.sink.input
        $self start
    }
}
```

```tcl
VstTcpListener $m2.listener \
    -port $port \
    -callback "$m2 listenerCallback"
```

Figure 7-5: Sample VuSystem program script fragment to create the server side of a TCP connection.

Figure 7-5 shows the program fragment that creates the server end of the TCP connection. A Tcl procedure `listenerCallback` is defined to set up a `VstTcpServer` module once it is created by the `VstTcpListener` module. The procedure first extracts any `-obj` keyword argument with the `keyarg` command and assign it to variable `obj`. This argument is the object command of the newly created `VstTcpServer` module. If `obj` is not the empty string, meaning there the `VstTcpServer` module is created, it will destroy the `VstTcpListener` module. Then it connects the media source’s output port `$self.source.output` to the Tcp server module’s input port `$obj.input`, and connects the Tcp server module’s output port `$obj.output` to the media sink module’s input port `$self.sink.input`. Finally, it start the in-band processing of the whole in-band module hierarchy by calling the `start` subcommand of the parent module of the module hierarchy. The program then creates a `VstTcpListener` module named `$m2.listener` with the Tcp port set to `$port` and the callback set to “`$m2 listenerCallback`”. When the `VstTcpListener` module receive a new tcp connection request on the TCP port specified, it will create a `VstTcpServer` module and evaluate the callback procedure with the `obj` keyword argument appended. The `obj` keyword argument is set to the object command of the `VstTcpServer` module created. Then the `listenerCallback` procedure will set up the actual Tcp connection.

Brent Philips developed a distributed programming extension [22] for the VuSystem. It provides modules for the connection of both the in-and and the out-of-band partition of the
<table>
<thead>
<tr>
<th>buffer size</th>
<th>TCP receive Mbits/second</th>
<th>TCP transmit Mbits/second</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5483</td>
<td>0.4037</td>
</tr>
<tr>
<td>2</td>
<td>0.7982</td>
<td>0.8149</td>
</tr>
<tr>
<td>4</td>
<td>1.1391</td>
<td>1.0712</td>
</tr>
<tr>
<td>8</td>
<td>1.2780</td>
<td>1.0481</td>
</tr>
<tr>
<td>16</td>
<td>1.2727</td>
<td>1.0561</td>
</tr>
<tr>
<td>32</td>
<td>1.2738</td>
<td>1.0733</td>
</tr>
<tr>
<td>64</td>
<td>1.2640</td>
<td>1.0871</td>
</tr>
<tr>
<td>128</td>
<td>1.2364</td>
<td>1.1276</td>
</tr>
<tr>
<td>256</td>
<td>1.2227</td>
<td>1.0950</td>
</tr>
<tr>
<td>512</td>
<td>1.2091</td>
<td>1.0463</td>
</tr>
<tr>
<td>1024</td>
<td>1.2069</td>
<td>1.0322</td>
</tr>
<tr>
<td>2028</td>
<td>1.2065</td>
<td>0.9610</td>
</tr>
<tr>
<td>4096</td>
<td>1.2880</td>
<td>0.9924</td>
</tr>
<tr>
<td>8172</td>
<td>1.2222</td>
<td>0.9498</td>
</tr>
<tr>
<td>16384</td>
<td>1.2094</td>
<td>1.0220</td>
</tr>
<tr>
<td>32768</td>
<td>1.2002</td>
<td>0.9741</td>
</tr>
</tbody>
</table>

Table 7.1: Wireless network TCP throughput measurement results.

application.

7.6 Performance Results

The raw TCP throughput of the WaveLAN wireless network system was measured.

Experimental Setup

The program used for the performance testing is ttcp, created by the US Army Ballistics Research Lab (BRL). It measures TCP network throughput with different send and receive buffer size. I measured the TCP transmit and receive throughput of different buffer sizes. The machines used to measure the WaveLAN throughput is the IBM ThinkPad 760C laptop computer.

Results

Table 7.1 shows the result of running ttcp with different buffer sizes. Figure 7-6 shows a plot of TCP throughput as a function of buffer size. The buffer size is plotted at log scale to emphasize the change in bandwidth at small buffer sizes. The throughput is low at very

\footnote{ttcp is available at ftp site ftp://sgi.com/sgi/src/ttcp.}
Figure 7-6: Plot of TCP network throughput as a function of buffer size.

small buffer size, probably because most of the bandwidth is used for TCP/IP headers. The bandwidth saturates at buffer size of over 10 bytes. The peak TCP receive throughput is about 1.25 Mbits per second. The peak TCP transmit throughput is about 1.06 Mbits per second.

The TCP transmit performance is lower than the TCP receive performance. The reason might be that the WavePOINT bridge can push data from the wired segment to the wireless segment faster than the WaveLAN network adaptor on the laptop can send data. It might also be caused by a better implementation of the receive code in the WaveLAN device driver.
Chapter 8

Integration and Evaluation

In this chapter, I evaluate how well the components in the mobile system integrate and perform together. Some applications of the mobile system are described in the next section. Then I will describe some integration problems I encountered. Overall performance results are presented in the last section.

8.1 Applications

The mobile system extends the reach of current VuSystem applications to mobile environments. This section describes some of the more interesting VuSystem applications that benefit from running on the mobile system.

8.1.1 Mobile Whiteboard Recorder

The Whiteboard Recorder was developed by William Stasior as an example computer-participative multimedia application using the VuSystem. By taking continue video from a stationary camera looking at a whiteboard, this application extracts the writings on the whiteboard and keeps a history of changes. It uses motion analysis to distinguish the person who writes on the whiteboard from the writings on the whiteboard. (Figure 8-1)

With the mobile system, users can use this application in meeting room without video equipment. He can also communicate the images of the writings through a low-bandwidth wireless network in which the transfer of a full motion video of a meeting would be infeasible and inefficient.
8.1.2 Remote control of Video Rover

The *Video Rover* was built by Vanu Bose to explore live video networking is the area of remote sensing and telepresence. The Video Rover is an untethered vehicle that communicates with the VuNet using wireless transmission. It consists of a video camera mounted on a small remote-controlled car and a wireless video link. Users can control the car from a VuSystem application running on a workstation on the VuNet, with the rover returning video feedback to driver of the vehicle. (Figure 8-2)

With the mobile system, the driver can control the Rover "on the move". I use X Window to remotely display the Rover application on the laptop so people can control the Rover through the wireless network while walking or moving around. No modification to the Rover application is needed, and all the computation (except the X Window server) runs on a high-speed workstation on the VuNet. This is a way of distributing computing resources according to the available computation and network resources on different machines.

8.2 Conflicts in Integration

There are some unexpected problems when the whole system is used. For example, when I use the mobile system to control the Video Rover through an X connection, the mouse click on the control panel is slowed down a lot. Both the mouse press and mouse release response
times are delayed a couple seconds, making the control of the Rover virtually impossible. Once the video framerate is turned down, the mouse clicks become responsive again. The reason is that when the Rover application uses the fastest frame rate to capture video images from the Rover camera, the images are pushed through the X connection faster than what the wireless network bandwidth allows. The video frames fill up the TCP send buffer quickly and blocks the VuSystem from sending additional packets. The Linux scheduler then puts the sending process to sleep for a period of time to allow the send buffer to clear. However, since the VuSystem is a single threaded application, the whole VuSystem is put to sleep. If at this point a mouse click event arrives, it has to wait in the X event queue until the VuSystem process wakes up, finishes sending the video frame, and then it will process the mouse client event.

8.3 Performance Results

This section measures the overall performance of the mobile system. I performed a few measurements that characterize the overall strength and bottlenecks of the system.

1. The VuSystem filter modules processing time test measures how fast the processing system can manipulate media data. This test indicates if the laptop computer has enough processing power for live manipulation of media data.
2. The *Total System Throughput* test measures performances of simple VuSystem applications that captures, process and output media data. This test gives good indication of how well the media I/O subsystem works with the processing system.

3. The *Video transmission throughput* and the Video transmission latency and jitter tests measure how well the mobile system transmit video stream through the wireless network. This test indicates how well the wireless network subsystem handle the demand of the rest of the subsystem.

The performance tests are described in the following sections.

### 8.4 VuSystem Filter Modules Processing Times

To verify that the laptop computer and mid-range PC workstations has enough processing power to process video stream with perceptual time granularity, I measured the processing time of two representative filter modules to process one video frame. The filter modules measured are the *VsPuzzle* module and the *VvEdge* modules. The *VsPuzzle* filter module scrambles video frames to form a video puzzle. The *VvEdge* filter module performs edge detection on video frames. Both filter modules perform substantial amount of computation and video processing and is a good representation of non-trivial filter modules.

#### 8.4.1 Experimental Setup

To measure the processing time of a filter module on one video frame, a VuSystem test program was written. Figure 8-3 shows the in-band processing modules in the program. *VsTestVideoSource* very cheaply generates a stream of video frame payloads for the program. The generated payloads are fed to the filter module, one of *VsPuzzle* or *VvEdge*, to be processed. The processed payloads then travel to the *VsRateMeter* module, which measures the frame rate of the video stream and are discarded by the *VsNullSink* module at the end of the processing pipeline.

As shown in Section 5.5.2, the processing time of the *VsTestVideoSource*, the *VsNullSink* and the *VsRateMeter* module are negligible when compared to a non-trivial filter module. Therefore the processing time of a video frame by the program is mostly contributed by the filter module. We can then approximate the filter processing time for one video frame to be 1/video frame rate.
Figure 8-3: The setup used to measure module processing time for the VsPuzzle and the VvEdge filter modules.

![Diagram](image)

Figure 8-4: A plot of module processing time per video frame as a function of frame size for the VsPuzzle and VvEdge filter modules.

I measured the filter processing time under an IBM ThinkPad 760C laptop, a Digital DECpc 450ST and a Digital DEC 3000/400. The DEC 3000/400 results are presented for performance comparison with a high-end workstation.

### 8.4.2 Results

Figure 8-4 shows the filter processing times of one frame as a function of video frame size. Note that the video frame size is plotted on a log scale because the number of pixels in a video frame increases as the square of the increase in frame size. For example, a 640x480 video frame contains four times as many pixels as a 320x240 video frame. Because the filter modules process video pixels, the filter processing time should be a linear function of the
<table>
<thead>
<tr>
<th>frame size</th>
<th>IBM ThinkPad 760C</th>
<th>Digital DECpc 450ST</th>
<th>Digital DEC 3000/400</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ms/frame</td>
<td>ns/pixel</td>
<td>ms/frame</td>
</tr>
<tr>
<td>VsPuzzle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>640x480</td>
<td>16.13</td>
<td>53</td>
<td>22.03</td>
</tr>
<tr>
<td>320x240</td>
<td>3.12</td>
<td>41</td>
<td>4.81</td>
</tr>
<tr>
<td>212x160</td>
<td>1.64</td>
<td>48</td>
<td>2.52</td>
</tr>
<tr>
<td>160x120</td>
<td>1.19</td>
<td>62</td>
<td>1.81</td>
</tr>
<tr>
<td>VvEdge</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>640x480</td>
<td>116.14</td>
<td>378</td>
<td>293.26</td>
</tr>
<tr>
<td>320x240</td>
<td>28.82</td>
<td>375</td>
<td>73.26</td>
</tr>
<tr>
<td>212x160</td>
<td>12.64</td>
<td>373</td>
<td>32.47</td>
</tr>
<tr>
<td>160x120</td>
<td>7.28</td>
<td>379</td>
<td>18.43</td>
</tr>
</tbody>
</table>

Table 8.1: Filter processing time of the VsPuzzle and VvEdge modules.

number of pixels in a frame.

Table 8.1 shows the filter processing time in milliseconds per frame and nanoseconds per pixel. On the IBM ThinkPad 760C laptop, it takes 3.12 millisecond for the VsPuzzle module and 28.82 millisecond for the VvEdge module to process one half size 320x240 pixel video frame.

On full-size 640x480 pixels frame, the IBM ThinkPad provides approximately 65% the performance of DEC 3000/400 on the computational intensive VvEdge module. On the memory intensive VsPuzzle module, the IBM ThinkPad provide 48% performance of DEC 3000/400. This shows that the memory speed of the ThinkPad 760C is relatively slow when compared to its processor speed.

The per pixel processing times of the VsPuzzle module vary a lot with the frame size. The module rearranges the pixels on the frame by a series of memcpy operations. The processing time per pixel goes down with increasing frame size from 160x120 pixels to 320x240 pixels because the setup time of the memcpy operation per pixel goes down. The processing time per pixel of 640x480 pixels frames goes up probably because the video frame size exceeds the size of the secondary cache on the machines.

8.5 Total System Throughput

To evaluate the performance of typical VuSystem applications running on the mobile system and PC workstations, I measured the throughput of two simple applications build around
Figure 8-5: Experimental Setup used to measure system throughput.

<table>
<thead>
<tr>
<th>frame size</th>
<th>IBM ThinkPad 760C frames/second</th>
<th>Digital DECpc 450ST frames/second</th>
</tr>
</thead>
<tbody>
<tr>
<td>VpPuzzle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>6</td>
<td>4.19</td>
</tr>
<tr>
<td>60x120</td>
<td>15</td>
<td>14.49</td>
</tr>
<tr>
<td>VpEdge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>6</td>
<td>3.61</td>
</tr>
<tr>
<td>160x120</td>
<td>15</td>
<td>14.49</td>
</tr>
</tbody>
</table>

Table 8.2: Measured frame rates of simple VuSystem applications using the VpPuzzle and VpEdge filter modules.

the VpPuzzle and the VpEdge modules.

8.5.1 Experimental Setup

Figure 8-5 shows the in-band processing modules in the application. It is similar to in-band modules setup used in measuring the filter module performance in the last section except the VpTestVideoSource module is replaced by a real video source module VpCardCamVideoSource and the VpNullSink module is replaced by an X Window sink VpWindowSink. The VpCardCamVideoSource module captures video frames from the CardCam video capture card and creates video frame payloads. The payloads are passed to a filter module (VpPuzzle or VpEdge) to be processed. The VpRateMeter module measures the video frame rate and finally the VpWindowSink display the video frames on the screen.

8.5.2 Results

Table 8.2 shows the frame rates of the two applications running on a ThinkPad 760C laptop and a DECpc 450ST desktop. The frame rates are very close to the VuSystem video capture frame rate without any filter measured in Section 5.5.2. Under 320x240 pixels frame size, the both applications achieve 6 frames per second throughput, while the throughput without the filters is 7.5 frames per second. Therefore, the time spent in the processing module is relatively small compared to the video capturing time. In fact
application throughput is constrained by the video capture hardware throughput at the 160x120 pixel frame size.

### 8.6 Video Transmission Throughput

A large class of multimedia applications require real-time transmission of video stream between hosts. I measured the performance of video transmission between the mobile host and a desktop workstation over a wireless network connection. I compared the results of video transmission with different video compression algorithms. This experiment gives can give a good measure of the wireless Ethernet performance in real world application. It also help us identify the tradeoffs between computation and network bandwidth.

Three different groups of experiments are performed:

1. Video transfer through an X Window remote display.
2. Video transfer through a TCP connection.
3. Two way video transfer through TCP connections.

#### 8.6.1 Experimental Setup

Figure 8-6 shows the setup for the X Window transfer experiment. The `VsVideoSource` module capture video frame payloads and handles to the `VsWindowSink` module. The `VsWindowSink` module displays the video frames on the X Window display. By configuring machine to use a remote X Window server, the X Window remote display facility will transfer the images over the network to the X display of another machine.
Figure 8.7: Experiential setup to measure video transfer throughput using TCP connection.

Figure 8.7 shows the in-band modules in the TCP video transmission test program. TCP connection is established by the the \texttt{VsTcpServer} and the \texttt{VsTcpClient} module pair. The \texttt{VsVideoSource} module captures video frames and create video frame payloads. The payloads are passed to the compression module to compress the video frame. Several compression modules are used in this experiment: the \texttt{VsJpegC} module performs JPEG compression, the \texttt{VsQRRC} module performs a quantized-run-length compression on black-and-white video frames, and the \texttt{VsCCCD} module performs Color Cell compression. The transparent filter \texttt{VsFilter} module is used if we don't want any compression. It just passes the payloads to the downstream module without processing. The compressed payloads are passed to the \texttt{VsTcpServer} module. It is the server side of the TCP connection. It converts the video payloads into byte streams and send it to the \texttt{VsTcpClient} module, the client side of the TCP connection.\footnote{The TCP connection is actually two-way. The \texttt{VsTcpServer} and the \texttt{VsTcpClient} modules can both send and receive payload from each other. However, in this program, only an one way connection is used.} The \texttt{VsTcpClient} converts the byte streams back to payloads. The decompression module then decompress the video frame back into \texttt{VuSystem} native video format. The decompression module should match the compression module used to compress the video frame;\footnote{If the compression scheme of the payload does not match the decompression module used, the payload will just pass to the downstream module without processing. Therefore, it is possible to construct an universal decompressor by connecting all the decompression modules together. It is not used in this experiment because it will affect the performance.} the \texttt{VsJpegD} module performs JPEG decompression, the \texttt{VsQRRLD} module performs quantized-run-length decompression, and the \texttt{VsCCCD} module performs Color Cell compression. Similarly, the transparent \texttt{VsFilter} module is used if this is what is used during compression.

The two-way video transmission experiment is set up by running two TCP transmission test program simultaneously, one transmits video from the local workstation to the mobile node, the other one transmits video from the mobile node back to the local station. Both
programs uses JPEG as a representative compression module to compress the video frames.

I used a Digital DEC 3000/400 workstation connected to a Vidboard as the local workstation in my experiments because I assume the local workstation should have plenty of processing and networking power, where as the laptop computer is the limiting factor in performance.

8.6.2 Results

Figure 8.3 shows the video transmission frame rates under different network connections with different compression schemes. From the results, I have made two observations:

First, video transmission performance from the DEC 3000/400 to the ThinkPad 760C is faster then from the ThinkPad 760C to the DEC 3000/400. This is because the Vidboard on the DEC 3000/4000 performs much faster then the CardCam video capture card on the ThinkPad. The second reason is because the ThinkPad is much slower then the DEC 3000/400, and it is more computationally expensive to do video capture and compression then to do video decompression and display. Therefore, the bottleneck is placed on the sender's side.

Second, by adjusting the quality of the compression scheme, video transmission frame rate increases with the compression ratio. Since the wireless network bandwidth is the bottleneck, we can trade computation resources with network bandwidth by compressing the video frame more before transmitting to achieve a higher frame rate. Note that however, how effective the extra compression is depends on the compression scheme. For example, with color JPEG compression, the ThinkPad to DEC TCP transmission frame rate actually went down comparing to the X Window connection. Also, with JPEG compression the frame rate goes up only slightly when the quality goes from 75 to 25 and the compression ratio goes up over 100%; With QRL compression, the frame rate goes up about 100% when the compression ratio goes up about 130%. This is because JPEG compression is much more computational intensive than QRL compression.

Table 8.4 shows the result of the two-way video transmission experiment. I used JPEG compression with the same quality on both ways as the representative compression scheme to evaluate the impact of a two-way transmission. As with the one-way video transmission experiment, there is an asymmetric transfer rate between transmitting from the DEC 3000/400 to the ThinkPad 760C and the other way around. This is again due to the dif-
<table>
<thead>
<tr>
<th>frame size</th>
<th>DEC 3000/400 to ThinkPad 760C</th>
<th>ThinkPad 760C to DEC 3000/400</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>frames per second</td>
<td>compression ratio</td>
</tr>
<tr>
<td>X Window connection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>160x120</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>TCP without compression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>1.6</td>
<td>1</td>
</tr>
<tr>
<td>160x120</td>
<td>7.6</td>
<td>1</td>
</tr>
<tr>
<td>TCP with JPEG compression (color)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quality: 75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>2.2</td>
<td>3</td>
</tr>
<tr>
<td>160x120</td>
<td>7.9</td>
<td>2</td>
</tr>
<tr>
<td>Quality: 50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>2.4</td>
<td>5</td>
</tr>
<tr>
<td>160x120</td>
<td>8.0</td>
<td>3</td>
</tr>
<tr>
<td>Quality: 25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>2.7</td>
<td>7</td>
</tr>
<tr>
<td>160x120</td>
<td>8.7</td>
<td>5</td>
</tr>
<tr>
<td>TCP with JPEG compression (Black and White)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quality: 75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>3.9</td>
<td>5</td>
</tr>
<tr>
<td>160x120</td>
<td>14.4</td>
<td>4</td>
</tr>
<tr>
<td>Quality: 50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>4.4</td>
<td>7</td>
</tr>
<tr>
<td>160x120</td>
<td>15.6</td>
<td>5</td>
</tr>
<tr>
<td>Quality: 25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>4.5</td>
<td>11</td>
</tr>
<tr>
<td>160x120</td>
<td>16.3</td>
<td>9</td>
</tr>
<tr>
<td>TCP with QRL compression (Black and White)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quality: 52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>3.2</td>
<td>2</td>
</tr>
<tr>
<td>160x120</td>
<td>6.7</td>
<td>2</td>
</tr>
<tr>
<td>Quality: 26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>6.3</td>
<td>4.6</td>
</tr>
<tr>
<td>160x120</td>
<td>10</td>
<td>3.7</td>
</tr>
<tr>
<td>TCP with CCC compression (color)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>320x240</td>
<td>7.1</td>
<td>4</td>
</tr>
<tr>
<td>160x120</td>
<td>20</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 8.3: Measured video transmission throughput using different connection and compression schemes.
Table 8.4: Measured two-way video transmission throughput with JPEG compression.

<table>
<thead>
<tr>
<th>Frame size</th>
<th>DEC 3000/400 to ThinkPad 760C frames/sec</th>
<th>ThinkPad 760C to DEC 3000/400 frames/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG color</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quality: 75</td>
<td>1.25</td>
<td>1</td>
</tr>
<tr>
<td>320x240</td>
<td>4.0</td>
<td>3.1</td>
</tr>
<tr>
<td>160x120</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>Quality: 25</td>
<td>4.9</td>
<td>3.4</td>
</tr>
</tbody>
</table>

| JPEG (Black and White) |                                          |                                          |
| Quality: 75            |                                          |                                          |
| 320x240                | 3.4                                      | 1.3                                      |
| 160x120                | 10                                       | 5                                        |
| Quality: 25            |                                          |                                          |
| 320x240                | 4.0                                       | 1.6                                      |
| 160x120                | 15                                       | 5.6                                      |

8.7 Video Transmission Latency and Jitter

With video transmission over the network, the video latency and jitter are expected to increase. This would affect the user-machine interaction in multimedia applications over the network. I measured the video latency and jitter over an X Window connection through the WaveLAN wireless network with different frame sizes and frame rates.

8.7.1 Experimental Setup

The VuSystem setup is the same as the setup used to measure the X Window connection throughput in last section (Figure 8-6). In addition, I instrumented the code to measure the latency between the time when a video frame is captured and the time when the TimeOut function in VsWindowSink is called to display the image on the remote screen. The network bottleneck in transmitting the video images will back-pressure and slow down the frame rate. Note that the measured latency does not include the time X Window spends on displaying
the image on the remote screen, which means that it does not include the network latency. The experiment is run on a ThinkPad 760C laptop with a WaveLAN network adaptor.

8.7.2 Results

Figure 8-8 and Figure 8-9 shows the video latencies through an X Window connection under 320x240 pixel and 160x120 pixel frame sizes. Two measurements are made for each frame size: a “fast” frame rate measurement in which the programs capture and displays frames at the fastest speed, and a slower frame rate measurement in which I reduced the frame rate to about half the fastest frame rate.

From the results shown in the figures, the jitter and latency is huge in the fast frame rate measurements. The VsWindowSink module keep on pushing frames on the remote X server. X Window uses TCP connections to transfer the frames. With the fast frame rate, the video frames will fill up the TCP output buffers quickly. This increases the latency because when the send buffer is full, the the VsWindowSink module need to wait extra time for TCP to accept the frame for sending. This also increases the jitter because TCP would send several frames in the send buffer at one time and causes period “bursts” of frames over the network. The bursts of frames are especially apparent in the 160x120 pixel frame size case. Because of the small frame size, the TCP send buffer can hold multiple frames and TCP sends more frames over the network at one time.

With lower frame rate the jitter improved a lot. There is not much variations in the latencies of frame. Also, the latencies in both the 320x240 pixel and the 160x120 pixel frame sizes are about the same. The reduced frame rate allows the VuSystem modules and TCP to have enough time between each frame to process and transfer one frame at a time.
Figure 8-8: Video transfer latency through X Window in 320x240 pixel mode.

Figure 8-9: Video transfer latency through X Window in 160x120 pixel mode.
Chapter 9

Conclusion

In this report, I have described the design of a mobile system for distributed multimedia applications. The mobile system integrates commercially available multimedia and networking equipment with a laptop computer. The software architecture will accommodate future extensions. This chapter concludes the report with a summary of work, a description of the lessons learned from this project, and ideas for future work.

9.1 Summary of Work

With increasing computing power, multimedia applications are playing more active role in the human-machine interactions. I have identified the need for multimedia systems to gain mobility and connectivity, and have developed a mobile station for distributed multimedia applications. Traditional multimedia systems rely on the processing power of desktop workstations attached to high-speed local area networks. Therefore multimedia applications are constrained to fixed locations and limited environments.

The mobile station has three different requirements: multimedia capability, mobility, and multimedia programming facilities. To satisfy these requirements, I have constructed a mobile system using commercially available mobile and multimedia hardware devices, combined with an extensible architecture that allow future extension as new hardware becomes available.

I used the VuSystem as the media processing system because of its programmability and extensibility. I have developed an extensible architecture that allow easy extension by adding media subsystems onto the processing system. There is a subsystem for each media
I/O function. It interfaces with the VuSystem through a VuSystem module.

I have constructed a video input subsystem, an audio input/output subsystem and a wireless network subsystem to provide the basic media I/O and communication functionality required by the mobile station. New subsystem can be added in the future to utilize newer devices or to provide new functionalities. In constructing the subsystems, I have developed a PCMCIA video capture device driver and several new VuSystem modules as the interfaces between the main processing system and the subsystems.

I have demonstrated that the mobile system has enough performance for video capture and media processing. The performance still falls short of desktop workstations, but the mobile system can utilize resources of other workstations through the wireless network. I have also found out that the video capture device presents a bottleneck in most applications. The wireless networking device also has severe latency and jitter problems under high network loads.

9.2 Lessons Learned

During the development of the mobile system, I have learned both the advantages and shortcomings of using a PC laptop computer to build a mobile multimedia system. Several lessons are learned:

1. It is feasible to construct a high performance multimedia mobile station with PC laptop computer and off-the-shelf components. Current high-end PC laptop computers provides enough processing power for dynamic manipulation of live media. PCMCIA expansion peripherals provides multimedia and wireless networking expansion. The mobile system runs simple multimedia applications with good performance. In fact, its raw processing power (as indicated by the VvEdge filter processing time test of Section 8.4) is surprising close to that of a workstation.

2. The bottlenecks of the mobile system are in the media input and output subsystems. The performance of the video capture subsystem is satisfactory given the small size of the PCMCIA video capture device. However, it is still too slow for high quality video applications. It is also the major bottleneck for multimedia applications because the computer can process video at much higher rates then the video capture rate.
3. The wireless network subsystem is another bottleneck for the mobile system. With limited bandwidth, media data has to be compressed to provide acceptable throughput. The system also lacks congestion management to deal with the varying quality of the network connection. The use of an inappropriate media compression scheme can result in large video jitter and latency, which can severely limit the user-machine interaction.

4. Scheduling problems arise as a result of interactions between the VuSystem, the X Window System, and the network software. As described in Section 8.2, (when we control the Video Rover through a remote X display with high frame rate) the Linux scheduler can block the VuSystem process, which prevents it from dealing with any X events, causing the mouse to be unresponsive. This problem arises because the VuSystem was designed as a single threaded application that implements its own resource sharing mechanism. However, the scheduler of the operating system schedule the VuSystem process as a whole, not individual tasks within the VuSystem.

9.3 Future Work

This section suggests possible future work to extend the mobile station. With the extensible architecture of the mobile station, subsystems can be added/upgraded independently. Most work is aimed at improving the shortcomings of the current system as described in the last section.

Rate Adaptive Video Compression Module

As shown in the video transmission test in Section 8.6, the wireless network presents a bottleneck for mobile video applications. Standard video compression algorithms help improving video transmission throughput, but they are not optimal for real-time operation in a wireless network environment where variation in bandwidth and latency is high. A rate adaptive video compression algorithm could be developed to change the compression parameters base on the quality/bandwidth of the connection. One example of such an algorithm is the H.261 video codec with control congestion scheme used in ivs [26] for video conferencing over low-bandwidth IP networks.
WaveLAN Roaming Support

Anthony Joseph is working on roaming support for the WaveLAN Linux device driver. We will also wire the whole Laboratory for Computer Science building with WavePOINT bridges in the near future. With roaming support, the mobile station will be able to switch among different WavePOINT bridges automatically when moving around the building. This would provide an ideal environment to study issues in wireless networking and to develop distributed mobile applications with multiple mobile hosts.

Video Rover II

The Video Rover described in Chapter 8.1.2 is a “dumb” robot capable of only transmitting analog video signals to a base station and receiving control signals from the driver. It would be interesting to integrate the mobile system with the Rover and make it capable of manipulating live video inputs. The new Rover could digest the input video and only transmit the interesting information. We can construct a stripped down version of the mobile system by including only the processing system into a small box with a single card CPU and two PCMCIA slots. (Figure 9-1) Users could then program the Rover to response to certain video events that it sees. Multiple Rovers could even communicate and interact with each other.
Appendix A

Modules In the VuSystem

This appendix describes three new VuSystem modules: the VsCardCamVideoSource, the VsLinuxAudioSource and the VsLinuxAudioSink modules. The VsCardCamVideoSource module is the interface between the processing system (Chapter 4) and the video input subsystem (Chapter 5). The VsLinuxAudioSource and the VsLinuxAudioSink modules are interfaces between the processing system and the audio I/O subsystem (Chapter 6). Please refer to [18] appendix A for a reference of other predefined modules in the VuSystem.

A.1 The VsCardCamVideoSource Module

The VsCardCamVideoSource module provides a video source interface to the Quadrant CardCam PCMCIA video capture card. It is based on the VsEntity module.

The VsCardCamVideoSource is usually created by the VsVideoSource module. The -videoSource : cardcam keyword parameter specifies the VsCardCamVideoSource module to be used in the VsVideoSource module.

The pathname VsCardCamVideoSource Subcommand

\[
<VsCardCamVideoSource> \text{pathname}\ [\langle\text{pathname}\rangle] \\
\Rightarrow \langle\text{pathname}\rangle
\]

The pathname VsCardCamVideoSource subcommand provides access to the device special file pathname for the CardCam. The default value is /dev/cardcam. It takes:

\[
\text{pathname (Pathname)} \text{ A new CardCam device file pathname.}
\]

It returns:

\[
\text{pathname (Pathname)} \text{ The current CardCam device file pathname.}
\]

The port VsCardCamVideoSource Subcommand

\[
<VsCardCamVideoSource> \text{port}\ [\langle\text{port}\rangle] \\
\Rightarrow \langle\text{port}\rangle
\]

The port VsCardCamVideoSource subcommand provides access to the input port for the CardCam. Port 1 is the composite video port. Port 2 is the S-video port. Port 3 generates a colorbar video test signal. It takes:
**port** *(svideo, colorbar, 1, 2, or 3)* A new input port.

It returns:

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

The **std** VsCardCamVideoSource Subcommand

```
<VsCardCamVideoSource> std [std]
```

The **std** VsCardCamVideoSource subcommand provides access to the video standard for the CardCam. It takes:

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

It returns:

**std** *(ntsc or pal)* The current video standard.

The **color** VsCardCamVideoSource Subcommand

```
<VsCardCamVideoSource> color [color]
```

The **color** VsCardCamVideoSource subcommand provides access to the color switch for the CardCam. 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 value.

The **brightness** VsCardCamVideoSource Subcommand

```
<VsCardCamVideoSource> brightness [brightness]
```

The **brightness** VsCardCamVideoSource subcommand provides access to the brightness adjustment for the CardCam, which ranges from -50 to 50 percent. It takes:

**brightness** *(Integer)* A new brightness.

It returns:

**brightness** *(Integer)* The current brightness.
The contrast \texttt{VsCardCamVideoSource Subcommand}

\begin{verbatim}
<VsCardCamVideoSource> contrast [\langle\text{contrast}\rangle]
  ==> \langle\text{contrast}\rangle
\end{verbatim}

The \texttt{contrast} \texttt{VsCardCamVideoSource subcommand} provides access to the contrast adjustment for the CardCam, which ranges from 0 to 200 percent. It takes:

\texttt{contrast (Integer) A new contrast.}

It returns:

\texttt{contrast (Integer) The current contrast.}

The saturation \texttt{VsCardCamVideoSource Subcommand}

\begin{verbatim}
<VsCardCamVideoSource> saturation [\langle\text{saturation}\rangle]
  ==> \langle\text{saturation}\rangle
\end{verbatim}

The \texttt{saturation} \texttt{VsCardCamVideoSource subcommand} provides access to the saturation adjustment for the CardCam, which ranges from 0 to 200 percent. It takes:

\texttt{saturation (Integer) A new saturation.}

It returns:

\texttt{saturation (Integer) The current saturation.}

The hue \texttt{VsCardCamVideoSource Subcommand}

\begin{verbatim}
<VsCardCamVideoSource> hue [\langle\text{hue}\rangle]
  ==> \langle\text{hue}\rangle
\end{verbatim}

The \texttt{hue} \texttt{VsCardCamVideoSource subcommand} provides access to the hue adjustment for the CardCam, which ranges from -45 to 45 degrees. It takes:

\texttt{hue (Integer) A new hue.}

It returns:

\texttt{hue (Integer) The current hue.}

The scale \texttt{VsCardCamVideoSource Subcommand}

\begin{verbatim}
<VsCardCamVideoSource> scale [\langle\text{scale}\rangle]
  ==> \langle\text{scale}\rangle
\end{verbatim}

The \texttt{scale} \texttt{VsCardCamVideoSource subcommand} provides access to the scale parameter for the CardCam. The scale parameter specifies the size of the video frames generated: 1 means full size 640x480 pixels, 2 means half size 320x240 pixels, and 4 means quarter size 160x120 pixels. It takes:

\texttt{scale (1, 2, or 4) A new scale.}

It returns:

\texttt{scale (1, 2, or 4) The current scale.}
The depth VsCardCamVideoSource Subcommand

\[ \langle VsCardCamVideoSource \rangle \text{ depth } [\langle \text{depth} \rangle] \]
\[ \text{==> } \langle \text{depth} \rangle \]

The depth VsCardCamVideoSource subcommand specifies the depth of the color video frames captured. It takes:

\text{depth } (8 \text{ or } 24) \text{ A new depth.}

It returns:

\text{depth } (8 \text{ or } 24) \text{ The current depth.}

The frameRate VsCardCamVideoSource Subcommand

\[ \langle VsCardCamVideoSource \rangle \text{ frameRate } [\langle \text{frameRate} \rangle] \]
\[ \text{==> } \langle \text{frameRate} \rangle \]

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

\text{frameRate } (\text{Integer}) \text{ A new frame rate.}

It returns:

\text{frameRate } (\text{Integer}) \text{ The current frame rate.}

The byteOrder VsCardCamVideoSource Subcommand

\[ \langle VsCardCamVideoSource \rangle \text{ byteOrder } [\langle \text{byteOrder} \rangle] \]
\[ \text{==> } \langle \text{byteOrder} \rangle \]

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

\text{byteOrder } (\text{msbFirst or lsbFirst}) \text{ A new byte order.}

It returns:

\text{byteOrder } (\text{msbFirst or lsbFirst}) \text{ The current byte order.}

The encoding VsCardCamVideoSource Subcommand

\[ \langle VsCardCamVideoSource \rangle \text{ encoding } [\langle \text{encoding} \rangle] \]
\[ \text{==> } \langle \text{encoding} \rangle \]

The encoding VsCardCamVideoSource subcommand specifies the color pixel encoding used for 24-bit color frames captured with the CardCam. It takes:

\text{encoding } (\text{bgr or rgb}) \text{ A new encoding.}

It returns:

\text{encoding } (\text{bgr or rgb}) \text{ The current encoding.}
A.2 The VsLinuxAudioSource Module

The VsLinuxAudioSource module provides an audio source interface to PC audio hardware under Linux. Depending on the capability of the audio mixer device, different PC audio hardware might not supports all the following module subcommands. The module subcommand will return TCL_ERROR if it is not supported. It is based on the VsEntity module.

The VsLinuxAudioSource is usually created by the VsAudioSource module. The -audioSource :linux keyword parameter specifies the VsLinuxAudioSource module to be used in the VsAudioSource module,

The **pathname VsLinuxAudioSource Subcommand**

```
<VsLinuxAudioSource> pathname [pathname]
  ==> <pathname>
```

The **pathname** VsLinuxAudioSource subcommand provides access to the the audio device node pathname parameter for a VsLinuxAudioSource module. The default value is /dev/audio. It takes:

`pathname (Pathname)` A new audio device file pathname.

It returns:

`pathname (Pathname)` The current audio device file pathname.

The **port VsLinuxAudioSource Subcommand**

```
<VsLinuxAudioSource> port [port]
  ==> <port>
```

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

`port (microphone, 1, 2, 3, or 4)` A new input audio.

It returns:

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

The **lineVol VsLinuxAudioSource Subcommand**

```
<VsLinuxAudioSource> lineVol [lineVol]
  ==> <lineVol>
```

The **lineVol** VsLinuxAudioSource subcommand provides access to the the volume level for the line in jack on the audio device. It takes:

`lineVol (Integer between 0 and 100)` A new volume level.

It returns:

`lineVol (Integer)` The current volume level.
The **micVol** VsLinuxAudioSource Subcommand

```text
<VsLinuxAudioSource> micVol [<micVol>]
  ==> <micVol>
```

The **micVol** VsLinuxAudioSource subcommand provides access to the the volume level for the microphone in jack on the audio device. It takes:

**micVol (Integer between 0 and 100)** A new volume level.

It returns:

**micVol (Integer)** The current volume level.

The **cdVol** VsLinuxAudioSource Subcommand

```text
<VsLinuxAudioSource> cdVol [<cdVol>]
  ==> <cdVol>
```

The **cdVol** VsLinuxAudioSource subcommand provides access to the the volume level for the CD audio input on the audio device. It takes:

**cdVol (Integer between 0 and 100)** A new volume level.

It returns:

**cdVol (Integer)** The current volume level.

The **imixVol** VsLinuxAudioSource Subcommand

```text
<VsLinuxAudioSource> imixVol [<imixVol>]
  ==> <imixVol>
```

The **imixVol** VsLinuxAudioSource subcommand provides access to the the output volume level for the recording monitor. It takes:

**imixVol (Integer between 0 and 100)** A new volume level.

It returns:

**imixVol (Integer)** The current volume level.

The **synthVol** VsLinuxAudioSource Subcommand

```text
<VsLinuxAudioSource> synthVol [<synthVol>]
  ==> <synthVol>
```

The **synthVol** VsLinuxAudioSource subcommand provides access to the the volume level for the build in synthesizer chip on the audio device. It takes:

**synthVol (Integer between 0 and 100)** A new volume level.

It returns:

**synthVol (Integer)** The current volume level.
The `recLevel` VsLinuxAudioSource Subcommand

```
<VsLinuxAudioSource> recLevel [<recLevel>]
```

It returns:

```
recLevel (Integer) The current volume level.
```

The `gain` VsLinuxAudioSource Subcommand

```
<VsLinuxAudioSource> gain [<gain>]
```

It returns:

```
gain (Integer) The current volume level.
```

A.3 The VsLinuxAudioSink Module

The VsLinuxAudioSink module provides an audio sink interface to PC audio hardware under Linux. Depending on the capability of the audio mixer device, different PC audio hardware might not support all the following module subcommands. The module subcommand will return TCL.ERROR if it is not supported. It is based on the VsEntity module.

The VsLinuxAudioSink is usually created by the VsAudioSink module. The -audioSink :linux keyword parameter specifies the VsLinuxAudioSink module to be used in the VsAudioSink module.

The `pathname` VsLinuxAudioSink Subcommand

```
<VsLinuxAudioSink> pathname [<pathname>]
```

It returns:

```
pathname (Pathname) The current audio device file pathname.
```

```
```

```
The \texttt{volume VsLinuxAudioSink} Subcommand

\begin{verbatim}
<VsLinuxAudioSink> volume [<volume>]
  ==><pathname>
\end{verbatim}

The \texttt{volume VsLinuxAudioSink} subcommand provides access to the the master output volume level (headphone/line out volume) on the audio device. It takes:

\texttt{volume (Integer between 0 and 100)} A new volume level.

It returns:

\texttt{volume (Integer)} The current volume level.

The \texttt{bass VsLinuxAudioSink} Subcommand

\begin{verbatim}
<VsLinuxAudioSink> bass [<bass>]
  ==><pathname>
\end{verbatim}

The \texttt{bass VsLinuxAudioSink} subcommand provides access to the base level on the audio device. It takes:

\texttt{bass (Integer between 0 and 100)} A new volume level.

It returns:

\texttt{bass (Integer)} The current volume level.

The \texttt{treble VsLinuxAudioSink} Subcommand

\begin{verbatim}
<VsLinuxAudioSink> treble [<treble>]
  ==><pathname>
\end{verbatim}

The \texttt{bass VsLinuxAudioSink} subcommand provides access to the bass level on the audio device. It takes:

\texttt{treble (Integer between 0 and 100)} A new volume level.

It returns:

\texttt{treble (Integer)} The current volume level.

The \texttt{pcm VsLinuxAudioSink} Subcommand

\begin{verbatim}
<VsLinuxAudioSink> pcm [<pcm>]
  ==><pathname>
\end{verbatim}

The \texttt{bass VsLinuxAudioSink} subcommand provides access to the output level for the digitized voice channel. It takes:

\texttt{pcm (Integer between 0 and 100)} A new volume level.

It returns:

\texttt{pcm (Integer)} The current volume level.
The speaker VsLinuxAudioSink Subcommand

\<VsLinuxAudioSink\> speaker [\<speaker>]
  \=> \<pathname>\

The **bass** VsLinuxAudioSink subcommand provides access to the output volume level for the PC speaker signals. It takes:

\textit{speaker} (*Integer between 0 and 100*) A new volume level.

It returns:

\textit{speaker} (*Integer*) The current volume level.

The gain VsLinuxAudioSink Subcommand

\<VsLinuxAudioSink\> gain [\<gain>]
  \=> \<pathname>\

The **gain** VsLinuxAudioSink subcommand provides access to the output volume level for the PC speaker signals. It takes:

\textit{gain} (*Integer between 0 and 100*) A new volume level.

It returns:

\textit{gain} (*Integer*) The current volume level.
Appendix B

The CardCam Device Driver

B.1 The cardcam.h Device Driver Header File

/*
 * $Id: cardcam.h,v 1.3 1996/05/23 03:40:55 pchan Exp $
 *
 * CardCam Video-IN device driver header file
 *
 */

#ifndef _CARDCAM_H
#define _CARDCAM_H

/* Address Map of CardCam Video-IN */

#define VDA ((FG->baseport+0x01)-1) /* BT812 Address Register */
#define VDC ((FG->baseport+0x03)-1) /* BT812 Control Register */
#define CTRL (FG->baseport+0x04) /* Board Control Register */
#define STAT (FG->baseport+0x06) /* Board I2C Register */
#define ODDF (FG->baseport+0x08) /* Odd Field (lower byte=Y, upper byte=C) */
#define EVEN (FG->baseport+0x0C) /* Even Field (lower byte=Y, upper byte=C) */

#define FRAME_EVEN 4
#define FRAME_ODD 6

/* BT812 Internal Registers */

#define R_INSELECT 0x00
#define R_OPMODE 0x04
#define R_INFORMAT 0x05
#define R_VTIMING 0x07
#define R_BRIGHT 0x08
#define R_CONTRAST 0x09
#define R_SATURATION 0x0A
#define R_HUE 0x0B

/* IOCTLs */

#define FG_SIZE_SET 0x00
#define FG_SIZE_GET 0x01
#define FG_STD_SET 0x02
#define FG_STD_GET 0x03
#define FG_PORT_SET 0x04
#define FG_PORT_GET 0x05
#define FG_OUTPUT_SET 0x06
#define FG_BRIGHT_SET 0x10
#define FG_BRIGHT_GET 0x11
#define FG_CONTRAST_SET 0x12
#define FG_CONTRAST_GET 0x13
#define FG_SAT_SET 0x14
#define FG_SAT_GET 0x15
#define FG_HUE_SET 0x16
/*
 * Color conversion
 * From YCbCr to RGB (Red, Green, Blue):
 * 
 *    R = Y + 1.402 (Cr - 0.34414) - 0.71414 (Cb - 0.772)
 *    G = Y
 *    B = Y + 1.772 (Cr)
 */

#define RED(Y, Cr, Cb) (((Y) + (Cr)*359/256) & 0xff)
#define GREEN(Y, Cr, Cb) (((Y) - (Cb)*88/256 - (Cr)*193/256) & 0xff)
#define BLUE(Y, Cr, Cb) (((Y) + (Cb)*454/256) & 0xff)

#endif

B.2 The cardcam_cs.c Device Driver Source File

/*
 * Id: cardcam_cs.c,v 1.11 1996/05/23 03:39:57 pchan Exp $
 * PCMCIA client driver for Quadrant CardCam Video-In video capture card
 * Patrick Chan, pchan@tns.lcs.mit.edu
 * Copyright 1996 Massachusetts Institute of Technology
 * Permission to use, copy, modify and distribute this software and its
 * documentation for using Quadrant CardCam video capture card is hereby
 * granted without fee, provided that the above copyright notice appear
 * in all copies and that both that copyright notice and this permission
 * notice appear in supporting documentation, and that the name of
 * M.I.T. not be used in advertising or publicity pertaining to
 * distribution of the software without specific, written prior
 * permission. M.I.T. makes no representations about the suitability of
 * this software for any purpose. It is provided "as is" without express
 * or implied warranty.
 * The code is based upon several other drivers:
 * Quadrant International's DOS CardCam driver.
 * Here is the copyright notice:
 *---------------------------------------------------------------
 * DosTV Copyright 1994 Quadrant Int'l
 * The following source code is unsupported and is intended as a
 * programming example only. Quadrant International makes no other
 * warranties or representations regarding the fitness of CardCam Video-In
 * or this source code for any purpose whatsoever. Quadrant
 * International assumes no liability for loss of data or the loss of
 * income, or expectations of income, that might derive from the use or
 * mis-use of this product and this source code.
 * Quadrant International will not be held liable for the criminal use of
 */
* its products.
*---------------------------------------------------------------
*    David Hinds' skeleton Linux PCMCIA example driver
*    Carlos Puchol's WinVision frame grabber device driver
* /

#include "config.h"
#include "k_compatible.h"

#ifdef MODULE
#define init_cardcam init_module
#endif

#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/ptrace.h>
#include <linux/malloc.h>
#include <linux/string.h>
#include <linux/_timer.h>
#include <asm/io.h>
#include <asm/system.h>
#include <asm/irq.h>

#include "version.h"
#include "cs_types.h"
#include "cs.h"
#include "cistpl.h"
#include "ds.h"
#include "cardcam.h"

/*
   All the PCMCIA modules use PCMCIA_DEBUG to control debugging. If
   you do not define PCMCIA_DEBUG at all, all the debug code will be
   left out. If you compile with PCMCIA_DEBUG=0, the debug code will
   be present but disabled -- but it can then be enabled for specific
   modules at load time with a 'pc_debug=$' option to lsmod.
*/

static int major_dev = 0;

#define MAJOR_NR major_dev
#define DEVICE_NAME "cardcam"

#ifdef PCMCIA_DEBUG
static int pc_debug = PCMCIA_DEBUG;
static char *version = "cardcam_cs.c 0.01 'n';
#endif

/* Size of memory window: 4K */
#define WINDOW_SIZE 0x1000

 plais===================================================================

/* Parameters that can be set with 'insmod' */
/* Bit map of interrupts to choose from */
static u_long irq_mask = 0xde98;
 plais===================================================================

/*
 The event() function is this driver's Card Services event handler.
 It will be called by Card Services when an appropriate card status
 event is received. The config() and release() entry points are
 used to configure or release a socket, in response to card insertion
 and ejection events. They are invoked from the skeleton event
 handler.
*/
static void cardcam_config(dev_link_t *link);
static void cardcam_release(u_long arg);
static int cardcam_event(event_t event, int priority,
et_callback_args_t *args);

/*
The attach() and detach() entry points are used to create and destroy
"instances" of the driver, where each instance represents everything
needed to manage one actual PCMCIA card.
*/
static dev_link_t *cardcam_attach(void);
static dev_link_t *cardcam_detach(dev_link_t *);

void cardcam_interrupt(int reg);
static void cardcam_SetVD(int reg, int val);
static char cardcam_GetVD(int reg);

/*
The dev_info variable is the "key" that is used to match up this
device driver with appropriate cards, through the card configuration
database.
*/
static dev_info_t dev_info = "cardcam_cs";

/*
A linked list of "instances" of the cardcam device. Each actual
PCMCIA card corresponds to one device instance, and is described
by one dev_link_t structure (defined in ds.h).
*/
static dev_link_t *dev_list = NULL;

/*
A dev_link_t structure has fields for most things that are needed
to keep track of a socket, but there will usually be some device
specific information that also needs to be kept track of. The
'priv' pointer in a dev_link_t structure can be used to point to
a device-specific private data structure, like this.
*/
typedef struct local_info_t {
  ioaddr_t baseport;
  u_int irq;
  caddr_t address;
  u_int width;
  u_int height;
  u_int pixels;
  u_int size;
  u_int std;
  u_char port;
  u_int output;
} local_info_t;
static local_info_t *FG = NULL;

static u_char tempbuf[8*NTSC_WIDTH];
static u_char redDitherTable[4][256];
static u_char greenDitherTable[4][256];
static u_char blueDitherTable[4][256];
static u_char ZAP[4][32768];
static CTRLNAME = 0x80;  /* 0x08=1/2, 0x0A=1/4, 0x00=full size */
static int ticks = 0;

/* Queue for the driver sleeping on the timer */
struct wait_queue * fg_timer_queue = NULL;

/*============================================*/
static int fg_open(struct inode *inode, struct file *file);
static int fg_read(struct inode *inode, struct file *file, char *buf,
       int count);
static int fg_ioctl(struct inode *inode, struct file *file,
       u_int cmd, u_long arg);
static void fg_close(struct inode *inode, struct file *file);
static void InitDitherTable(u_char table[4][256], int scale, int mul, int add);
static void InitZAPTable(void);
static int Red(int y, int cr, int cb);
static int Green(int y, int cr, int cb);
static int Blue(int y, int cr, int cb);

static struct file_operations fg_fops = {
    NULL,      /* lseek */
    fg_read,   /* read */
    NULL,      /* write */
    NULL,      /* readdir */
    NULL,      /* select */
    fg_ioctl,  /* ioctl */
    NULL,      /* mmap */
    fg_open,   /* open */
    fg_close,  /* close */
    NULL,      /* fsync */
};

static u_char VDECODER_WTSC[] =
{ 0x28, 0x00, 0x00, 0xC0, /* 0-3 */
  0x00, 0x00, 0x00, 0x3C, /* 4-7 */
  0x00, 0x80, 0x80, 0x00, /* 8-B */
  0x0C, 0x03, 0x75, 0x00, /* C-F */
  0x80, 0x02, 0x16, 0x00, /* 10-13 */
  0x80, 0x01, 0x4B, 0x00, /* 14-17 */
  0x12, 0x5A, 0x46, 0x00, /* 18-1B */
  0x00, /* 1C */
};

static u_char VDECODER_PAL[] =
{ 0x28, 0x00, 0x00, 0xC0, /* 0-3 */
  0x00, 0x74, 0x40, 0x00, /* 4-7 */
  0x00, 0x80, 0x80, 0x00, /* 8-B */
  0x80, 0x03, 0x9B, 0x00, /* C-F */
  0x00, 0x03, 0x21, 0x00, /* 10-13 */
  0x3E, 0x02, 0xC6, 0x3C, /* 14-17 */
  0x13, 0x6C, 0x56, 0x00, /* 18-1B */
  0x00, /* 1C */
};

static u_char * VDECODER_WTSC = VDECODER_WTSC;
static struct wait_queue *fg_wait_queue = 0;

阊卡 dormbcdl_rea山) creates an "instance" of the driver, allocating
local data structures for one device. The device is registered
with Card Services.

The dev_link structure is initialized, but we don't actually
configure the card at this point -- we wait until we receive a
card insertion event.
static dev_link_t *cardcam_attach(void)
{
    client_reg_t client_reg;
    dev_link_t *link;
    local_info_t *local;
    int ret;

    ifndef PCMCIA_DEBUG
    if (pc_debug)
        printk("cardcam_cs: cardcam_attach()\n");
    endif

    /* Initialize the dev_link_t structure */
    link = kmalloc(sizeof(struct dev_link_t), GFP_KERNEL);
    memset(link, 0, sizeof(struct dev_link_t));
    link->release.function = &cardcam_release;
    link->release.data = (u_long)link;

    /* The io structure describes IO port mapping */
    link->io.NumPorts1 = 16;
    link->io.Attributes1 = IO_DATA_PATH_WIDTH_AUTO;
    link->io.IOAddrLines = 5;

    /* Interrupt setup */
    link->irq.Attributes = IRQ_TYPE_EXCLUSIVE;
    link->irq IRQInfo1 = IRQ_IRQ02_VALID|IRQ_LEVEL_ID;
    link->irq IRQInfo2 = irq_mask;
    link->irq Handler = cardcam_interrupt;

    /* General socket configuration */
    link->conf.Attributes = CONF_ENABLE_IRQ;
    link->conf.Vcc = 50;
    link->conf.IntType = INT_MEMORY_AND_IO;
    link->conf.ConfigIndex = 1;
    link->conf.Present = PRESENT_OPTION;

    /* Allocate space for private device-specific data */
    local = kmalloc(sizeof(local_info_t), GFP_KERNEL);
    memset(local, 0, sizeof(local_info_t));
    link->priv = local;

    local->width = NTSC_WIDTH/2;
    local->height = NTSC_HEIGHT/2;
    local->pixels = local->width * local->height;
    local->size = 2;
    local->port = 0;
    local->std = STD_NTSC;
    local->output = OUTPUT_COLOR;
    FG = local;

    /* Register with Card Services */
    link->next = dev_list;
    dev_list = link;
    client_reg.dev_info = &dev_info;
    client_reg.Attributes = INFO_IO_CLIENT | INFO_CARD_SHARE;
    client_reg.EventMask =
    CS_EVENT_CARD_INSERTION | CS_EVENT_CARD_REMOVAL |
    CS_EVENT_RESET_PHYSICAL | CS_EVENT_CARD_RESET |
    CS_EVENT_PM_SUSPEND | CS_EVENT_PM_RESUME;
    client_reg.event_handler = &cardcam_event;
    client_reg.Version = 0x0010;
    client_reg.event_callback.args.client_data = link;
    ret = CardServices(RegisteClient, &link->handle, &client_reg);
    if (ret != 0) {
        cs_error(RegisteClient, ret);
        cardcam_detach(link);
        return NULL;
    }

    return link;
}
} /* cardcam_attach */

/*===========================================================================

This deletes a driver "instance". The device is de-registered with Card Services. If it has been released, all local data structures are freed. Otherwise, the structures will be freed when the device is released.

===========================================================================*/

static void cardcam_detach(dev_link_t *link)
{
    dev_link_t **linkp;

    ifdef PCMCIA_DEBUG
        if (pc_debug)
            printk("cardcam_cs: cardcam_detach(0x%p)\n", link);
    endif

    /* Locate device structure */
    for (linkp = &dev_list; *linkp; linkp = &(*linkp)->next)
        if (*linkp == link) break;
    if (*linkp == NULL)
        return;

    /* If the device is currently configured and active, we won't actually delete it yet. Instead, it is marked so that when the release() function is called, that will trigger a proper detach(). */
    if (link->state & DEV_CONFIG) {
        printk("cardcam_cs: detach postponed, '%s' still locked\n", link->dev_name);
        link->state |= DEV_STALE_LINK;
        return;
    }

    /* Break the link with Card Services */
    if (link->handle)
        CardServices(DeregisterClient, link->handle);

    /* Unlink device structure, free pieces */
    *linkp = link->next;
    if (link->priv) {
        kfree_s(link->priv, sizeof(local_info_t));
    }
    kfree_s(link, sizeof(struct dev_link_t));
}
} /* cardcam_detach */

/*===========================================================================

cardcam_config() is scheduled to run after a CARD_INSERTION event is received, to configure the PCMCIA socket, and to make the video capture device available to the system.

===========================================================================*/

static void cardcam_config(dev_link_t *link)
{
    local_info_t *dev;
    client_handle_t handle;
    win_req_t req;
    int i, j;
    modwin_t mod;
    memreq_t mem;

    handle = link->handle;

    ifdef PCMCIA_DEBUG

113
if (pc_debug)
    printk("cardcam_cs: cardcam_config(0x%p)\n", link);
#endif

    /* Configure card */
    link->state |= DEV_CONFIG;

    do {
        /* Allocate an IO port window */
        for (j = 0x300; j < 0x400; j += 0x20) {
            link->io.BasePort1 = j;
            /*link->io.BasePort1 = 0x380;*/
            i = CardServices(RequestIO, link->handle, &link->io);
            if (i == CS_SUCCESS) break;
        }
        if (i != CS_SUCCESS) {
            cs_error(RequestIO, i);
            break;
        }
    }
#endif

    /* Now allocate an interrupt line. Note that this does not
     * actually assign a handler to the interrupt.
     */
    i = CardServices(RequestIRQ, link->handle, &link->irq);
    if (i != CS_SUCCESS) {
        cs_error(RequestIRQ, i);
        break;
    }
#endif

    /* This actually configures the PCMCIA socket -- setting up
     * the I/O windows and the interrupt mapping.
     */
    i = CardServices(RequestConfiguration, link->handle, &link->conf);
    if (i != CS_SUCCESS) {
        cs_error(RequestConfiguration, i);
        break;
    }

    /* Allocate a memory window. Note that the dev_link_t
     * structure provides space for one window handle -- if your
     * device needs several windows, you'll need to keep track of
     * the handles in your private data structure, link->priv.
     */
    req.Attributes = WIN_DATA_WIDTH_16|WIN_MEMORY_TYPE_AM|WIN_ENABLE;
    req.Base = NULL;
    req.Size = WINDOW_SIZE;
    req.AccessSpeed = 0;
    link->win = (window_handle_t)link->handle;
    i = CardServices(RequestWindow, &link->win, &req);
    if (i != 0) {
        cs_error(RequestWindow, i);
        break;
    }
#endif

    if (pc_debug)
printk("cardcam_cs: cardcam_config: RequestWindow (Address: 0x%x)\n", (int)req.Base);
#endif

mod.Attributes = WIN_DATA_WIDTH_16|WIN_MEMORY_TYPE_CM|WIN_ENABLE;
i = Cardservices(ModifyWindow, link->win, &mod);
if (i != 0) {
cs_error(ModifyWindow, i);
break;
}
mem.CardOffset = 0x0000;
mem.Page = 0;
i = Cardservices(MapMemPage, link->win, &mem);
if (i != 0) {
cs_error(MapMemPage, i);
break;
}
} while (0);

dev = (local_info_t *)link->priv;
dev->baseport = link->io.BasePort1;
dev->irq = link->irq.AssignedIRQ;
dev->address = req.Base;
sprintf(link->dev_name, "cardcam");
link->major = major_dev;
link->minor = 0;

InitDitherTable(redDitherTable, 6, 9, 5, 4, 0);
InitDitherTable(greenDitherTable, 9, 4, 0);
InitDitherTable(blueDitherTable, 4, 1, 16);
InitZAPTable();

link->state &= ~DEV_CONFIG_PENDING;
/* If any step failed, release any partially configured state */
if (i != 0) {
cardcam_release(u_long)link);
return;
}
} /* cardcam_config */

/**************************************************************

After a card is removed, cardcam_release() will unregister the net
device, and release the PCMCIA configuration. If the device is
still open, this will be postponed until it is closed.

***************************************************************/

static void cardcam_release(u_long arg)
{
    dev_link_t *link = (dev_link_t *)arg;

#ifdef PCMCIA_DEBUG
    if (pc_debug)
        printk("cardcam_cs: cardcam_release(0x%p)\n", link);
#endif

    /* If the device is currently in use, we won't release until it
       is actually closed. */
    if (link->open) {
        printk("cardcam_cs: release postponed, '%s' still open\n",
               link->dev_name);
        link->state |= DEV_STALE_CONFIG;
        return;
    } /* Don't bother checking to see if these succeed or not */
    Cardservices(ReleaseWindow, link->win);
    Cardservices(ReleaseConfiguration, link->handle);
CardServices_ReleaseIO, link->handle, &link->io);
CardServices_ReleaseIRQ, link->handle, &link->irq);
link->state &= "DEV_CONFIG; 

if (link->state & DEV_STALE_LINK)
    cardcam_detach(link);
} /* cardcam_release */

The card status event handler. Mostly, this schedules other stuff to run after an event is received. A CARD_REMOVAL event also sets some flags to discourage the net drivers from trying to talk to the card any more.

When a CARD_REMOVAL event is received, we immediately set a flag to block future accesses to this device. All the functions that actually access the device should check this flag to make sure the card is still present.

static int cardcam_event(event_t event, int priority,
        event_callback_args_t *args)
{
    dev_link_t *link = args->client_data;

    /* ifdef PCMCIAD_DEBUG
        if (pc_debug)
            printk("cardcam_cs: cardcam_event()\n");
    */

    switch (event) {
    /* ifdef PCMCIAD_DEBUG
      case CS_EVENT_REGISTRATION_COMPLETE:
        if (pc_debug)
            printk("cardcam_cs: registration complete\n");
      */
      case CS_EVENT_CARD_REMOVAL:
        link->state &= "DEV_PRESENT; 
        if (link->state & DEV_CONFIG) {
            "/*((local_info_t *)link->priv)->block = 1;*/
            link->release.expires = ROW_AT(S); 
            add_timer(&link->release);
        } 
        break;
    case CS_EVENT_CARD_INSERTION:
        link->state = "DEV_PRESENT" | DEV_CONFIG_PENDING; 
        cardcam_config(link);
        break;
    case CS_EVENT_PM_SUSPEND:
        link->state |= DEV_SUSPENDED; 
        /* Fall through... */
        case CS_EVENT_RESET_PHYSICAL:
        if (link->state & DEV_CONFIG) 
            CardServices_ReleaseConfiguation, link->handle);
        break;
    case CS_EVENT_PM_RESUME:
        link->state &= "DEV_SUSPENDED; 
        /* Fall through... */
    case CS_EVENT_CARD_RESET:
        if (link->state & DEV_CONFIG) 
            CardServices_RequestConfiguration, link->handle, &link->conf);
        break;
    } 
    return 0;
} /* cardcam_event */

;?>
static unsigned long tvtojiffies(struct timeval *value)
{
    return((unsigned long)value->tv_sec * HZ +
    (unsigned long)(value->tv_usec + (1000000 / HZ - 1)) /
    (1000000 / HZ));
}

static void fg_wake (unsigned long ignored)
{
    /* The timer has just expired */
    wake_up(&fg_timer_queue);
}

static void fg_sleep( unsigned int ms)
{
    struct timeval millisecs;
    static struct timer_list fg_timer = { NULL, NULL, 0, 0, fg_wake };

    millisecs.tv_sec = 0;
    millisecs.tv_usec = ms * 1000 ;
    del_timer(&fg_timer);
    if (ms) {
        fg_timer.expires = tvtojiffies(&millisecs);
        add_timer(&fg_timer);
        sleep_on(&fg_timer_queue);
    }
}

void cardcam_interrupt(int reg)
{
    int safe;

#define PCMCIA_DEBUG
    if (pc_debug > 1)
        printk("cardcam_cs: interrupt\n");
#else
    ticks++;
    for (safe=0;safe<256;safe++) inw(CTRL);

    outw(0x00|CTRLMASK, CTRL);
    outw(0x10|CTRLMASK, CTRL);

    wake_up_interruptible(&fg_wait_queue);
#endif

} /* cardcam_interrupt */

/*====================================================================*/

int init_cardcam(void)
{
    servinfo_t serv;
    int i;
#ifndef PCMCIA_DEBUG
    if (pc_debug)
        printk(version);
#endif
    CardServices(GetCardServicesInfo, &serv);
    if (serv.Revision != CS_RELEASE_CODE) {
        printk("cardcam: Card Services release does not match!\n");
        return -1;
    }
    register_pcmcia_driver(&dev_info, &cardcam_attach, &cardcam_detach);

    /* Set up character device for user mode clients */
    i = register_chrdev(0, "cardcam", &fg_fops);
    if (i == -EBUSY)
        printk("unable to find a free device \$ for CardCam\n");
    else
        major_dev = i;

    return 0;
}
#ifdef MODULE
void cleanup_module(void)
{
    printk("cardcam_cs: unloading\n");
    unregister_pcmcia_driver(&dev_info);
    if (major_dev != 0)
        unregister_chrdev(major_dev, "cardcam");
    while (dev_list != NULL)
    {
        if (dev_list->state & DEV_CONFIG)
            cardcam_release((u_long)dev_list);
        cardcam_detach(dev_list);
    }
}
#endif /* MODULE */

static void InitDitherTable(u_char table[4][256], int scale, int mul, int add)
{
    u_int i, j;
    int dither, val;
    for (j = 0; j < 4; j++)
    {
        for (i = 0; i < 256; i++)
        {
            dither = (((j+1)%4)*256)/(scale+4);
            val = i+dither;
            if (val < 0) val = 0;
            if (val > scale) val = scale-1;
            table[j][i] = val*mul + add;
        }
    }
}

static int Red(int y, int cr, int cb)
{
    int i;
    i = y + cr*359/256;
    if (i>255) i=255;
    else if (i<0) i=0;
    return i;
}

static int Green(int y, int cr, int cb)
{
    int i;
    i = y - cb*88/256 - cr*183/256;
    if (i>255) i=255;
    else if (i<0) i=0;
    return i;
}

static int Blue(int y, int cr, int cb)
{
    int i;
    i = y + cb*454/256;
    if (i>255) i=255;
    else if (i<0) i=0;
    return i;
}

static void InitZAPTTable(void)
{
    int i, y, u, v, index, k;
    for (i=0; i<4; i++)
    {
        u_char * r = &redDitherTable[i][0];
        u_char * g = &greenDitherTable[i][0];
        u_char * b = &blueDitherTable[i][0];
        for (y=0; y<255; y+=8)
        {
            for (v=0; v<128; v+=8)
            {
                index=((y<7)&0x7E)|(v<3)&0x0E);
                k = (x[Red(y,v,u)]+ g[Green(y,v,u)]+ b[Blue(y,v,u)])/
                    (y<255 && v<128);  // Fix the division by zero issue
            }
        }
    }
}

118
else if (k<0) k=0;
ZAP[i][index]=k;
}
}
}

static int fg_open(struct inode *inode, struct file *file) {
int i;

#ifdef PCMCIA_DEBUG
if (pc_debug) {
    printk("cardcam_cs: cardcam_open()\n");
    printk(" dev minor=%d\n", MINOR(inode->i_rdev));
    printk(" dev major=%d\n", MAJOR(inode->i_rdev));
}
#endif
outb(0x80, STAT); /* hardware reset */
inw(STAT);
inw(STAT);
fg_sleep(5);

for (i=0; i<0x10; i++) {
    cardcam_SetVD(i, VDECORDER[i]);
}

#ifdef PCMCIA_DEBUG
if (pc_debug > 5) {
    printk("cardcam_cs: cardcam_open() success\n");
}
#endif

return 0;
}

void install(void) {
#ifdef PCMCIA_DEBUG
if (pc_debug > 5) {
    printk("cardcam_cs: install()\n");
}
#endif
outw(CTRLMASK|0x00, CTRL);
inw(CTRL);
outw(CTRLMASK|0x10, CTRL);
}

void iremove(void) {
#ifdef PCMCIA_DEBUG
if (pc_debug > 5) {
    printk("cardcam_cs: iremove()\n");
}
#endif
outw(CTRLMASK|0x00, CTRL);
ticks=0;
}

static int fg_read(struct inode *inode, struct file *file, char *buf,
        int count) {
    int x, y, row, col;
u_short * cmdbuf;
volatile u_short loop;
char *p1 = buf, *p2;
    u_char cy1, cy2, cr, cb;

119
u_short *sp1 = (u_short *)buf, *sp2;

cambuf = (u_short *)(FG->address + 8);

#ifdef PCMCIA_DEBUG
    if (pc_debug) {
        printk("cardcam_cs: fg_read()\n");
        printk("  cambuf = 0x%x, CTRL = 0x%x\n", (int)cambuf, CTRL);
    }
#endif

outw(0x00,CTRL);

outw(0x40,CTRL); /* reset address counter */
ins(VDA);
ins(VDA);
ins(VDA);
outw(0x00,CTRL);

/* Capture a frame */
outw(0x00,CTRL);
ins(VDA);
ins(VDA);
outw(0x80,CTRL);
cardcam_SetVD(R_VTIMING,0x80);

/**** install interrupt ****/
install();
interruptible_sleep_on(&fg_wait_queue);
if (current->signal & ~current->blocked) return -EINTR;
if (cardcam_GetVD(R_VTIMING) & 0x10) {
    interruptible_sleep_on(&fg_wait_queue);
    if (current->signal & ~current->blocked) return -EINTR;
}

#ifdef PCMCIA_DEBUG
    if (pc_debug > 0)
        printk("cardcam_cs: ticks %d\n", ticks);
#endif

iremove();

/****

cardcam_SetVD(R_VTIMING,0x00);

ins(ODDF);
ins(EVEN);

#ifdef PCMCIA_DEBUG
    if (pc_debug > 5)
        printk("cardcam_cs: fg_read: init success\n");
#endif

switch (FG->output) {
    case OUTPUT_RAW:
        row = count/2/FG->width;
col = FG->width;
        for(y=0; y<row; y++) {
            sp2 = (u_short *)tempbuf;
            for (x=0; x<col; x++) {
                *sp2++ = (u_short *)cambuf;
            }
            memcpy(tofs, p1, tempbuf, col*sizeof(u_short));
            sp1 += col;
        }
        break;
    case OUTPUT_COLOR:
        row = count/FG->width;
col = FG->width / 2;
        for(y=0; y<row; y++) {
p2 = tempbuf;
for (x=0;x<col;x++){
    loop=cambuf;
    cr=((loop>>8)&0xff)-128;
    cy1 =((loop)&0xff);
    loop=cambuf;
    cb=((loop>>8)&0xff)-128;
    cy2 =((loop)&0xff);

    *p2++ = ZAP[(yX2)*2]
        [((cy1<<7)&0x7C00)|((cr<<2)&0x3E0)|((cb>>3)&0x001F)];
    *p2++ = ZAP[(yX2)*2+1]
        [((cy2<<7)&0x7C00)|((cr<<2)&0x3E0)|((cb>>3)&0x001F)];
    }
    memcpy_tofs(p1, tempbuf, col*2);
    p1 += col*2;
}
break;

case OUTPUT_BGR:
    row = count/4/FG->width;
    col = FG->width / 2;
    for(y=0; y<row; y++) { 
        p2 = tempbuf;
        for (x=0;x<col;x++){
            loop=cambuf;
            cr=((loop>>8)&0xff)-128;
            cy1 =loop&0xff;
            loop=cambuf;
            cb=((loop>>8)&0xff)-128;
            cy2 =loop&0xff;

            *p2++ = BLUE(cy1,cr,cb);
            *p2++ = GREEN(cy1,cr,cb);
            *p2++ = RED(cy1,cr,cb);
P2++;
            *p2++ = BLUE(cy2,cr,cb);
            *p2++ = GREEN(cy2,cr,cb);
            *p2++ = RED(cy2,cr,cb);
P2++;
        }
        memcpy_tofs(p1, tempbuf, col*8);
        p1 += col*8;
    }
break;

case OUTPUT_GRAY:
    row = count / FG->width;
    col = FG->width;
    for(y=0; y<row; y++) {
        p2 = tempbuf;
        for (x=0;x<col;x++){
            loop=cambuf;
            cy1=loop&0xff;
            if (cy1 < 16) loop = 16; else if (cy1 > 239) cy1 = 239;
            *p2++ = cy1;
        }
        memcpy_tofs(p1, tempbuf, col);
        p1 += col;
    }
break;

case OUTPUT_NULL:
    row = count/2/FG->width;
    col = FG->width;
    for(y=0; y<row; y++) {
        memcpy_tofs(p1, tempbuf, col*sizeof(u_short));
        p1 += col;
    }
break;

default:
return(-1);
}

#ifdef PCMCIA_DEBUG
def pc_read()
\textbf{if} (pc_debug > 5)
printk("cardcam_cs: fg_read: captured %i bytes\n", sizeof(buf));
#endif

return count;
}

static int fg_ioctl(struct inode *inode, struct file *file, u_int cmd, u_long arg)
{
    int x, rtn=0;
    char c;

#ifdef PCMCIA_DEBUG
    if (pc_debug)
        printk("cardcam_cs: fg_ioctl(%u\n", cmd);
#endif

switch (cmd) {
    case FG_SIZE_SET:
        if ((u_char)arg == 1) {
            FG->size = 1;
            FG->width = MTSC_WIDTH;
            FG->height = MTSC_HEIGHT;
            FG->pixels = FG->width * FG->height;
            CTRLMASK = 0x00;
        } else if ((u_char)arg == 2) {
            FG->size = 2;
            FG->width = MTSC_WIDTH/2;
            FG->height = MTSC_HEIGHT/2;
            FG->pixels = FG->width * FG->height;
            CTRLMASK = 0x80;
        } else if ((u_char)arg == 4) {
            FG->size = 4;
            FG->width = MTSC_WIDTH/4;
            FG->height = MTSC_HEIGHT/4;
            FG->pixels = FG->width * FG->height;
            CTRLMASK = 0xA0
        } else {
            rtn = -EINVAL;
        }
        break;
    case FG_BRIGHT_SET:
        cardcam_SetVD(R_BRIGHT, ((int)arg&0x7f)<<1);
#ifdef PCMCIA_DEBUG
        if (pc_debug)
            printk("cardcam_cs: FG_BRIGHT_SET(%i)\n", (int)arg);
#endif
        inv(VDA);
        inv(VDA);
        inv(VDA);
        break;
    case FG_BRIGHT_GET:
        c=(cardcam_GetVD(R_BRIGHT)&0xFF)>>1;
        put_user_byte(c, (char *) arg);
        break;
    case FG_CONTRAST_SET:
        cardcam_SetVD(R_CONTRAST, ((int)arg&0x7f)<<1);
#ifdef PCMCIA_DEBUG
        if (pc_debug)
            printk("cardcam_cs: FG_CONTRAST_SET(%i)\n", (int)arg);
#endif
        inv(VDA);
        inv(VDA);
        inv(VDA);
        break;
}
case FG_CONTRAST_SET:
    put_user_byte(cardcam_GetVD(R_CONTRAST)>>1, (char *) arg);
    break;

case FG_SAT_SET:
    cardcam_SetVD(R_SATURATION, ((int)arg&0x7f)<<1);
    if (pc_debug)
        printk("cardcam_cs: FG_SAT_SET(%d)\n", (int)arg);
    break;

case FG_HUE_SET:
    cardcam_SetVD(R_HUE, ((int)arg&0x7f)<<1);
    if (pc_debug)
        printk("cardcam_cs: FG_HUE_SET(%d)\n", (int)arg);
    break;

case FG_HUE_GET:
    put_user_byte(cardcam_GetVD(R_HUE)>>1, (char *) arg);
    break;

case FG_PORT_SET:
    if (pc_debug)
        printk("cardcam_cs: FG_PORT_SET(%d)\n", (int)arg);
    break;

    if ((u_char)arg == PORT_COMPOSITE) {
    FG->port = (u_char)arg;
    outb(0x80, STAT);
    inw(STAT);
    for (x=0;x<0x10;x++) {
        cardcam_SetVD(x,VDECODER[x]);
    }
    cardcam_SetVD(R_INSELECT, 0x28);
    cardcam_SetVD(R_INFORMAT, 0x00);
    inw(VDA);
    inw(VDA);
    inw(VDA);
    }
    else if ((u_char)arg == PORT_SVIDEO) {
    FG->port = (u_char)arg;
    outb(0x80, STAT);
    inw(STAT);
    for (x=0;x<0x10;x++) {
        cardcam_SetVD(x,VDECODER[x]);
    }
    cardcam_SetVD(R_INSELECT, 0x00);
    cardcam_SetVD(R_INFORMAT, 0x80);
    inw(VDA);
    inw(VDA);
    inw(VDA);
    }
    else if ((u_char)arg == PORT_COLORBAR) {
    FG->port = (u_char)arg;
    outb(0x80, STAT);
    inw(STAT);
    for (x=0;x<0x10;x++) {
        cardcam_SetVD(x,VDECODER[x]);
    }
    cardcam_SetVD(R_OPMODE, 0x40);
    inw(VDA);
    inw(VDA);
    inw(VDA);
    }
    else {
        rtn = -EINVAL;
    }
break;
case FG_PORT_GET:
    put_user_byte(FG->port, (u_char *) arg);
    break;
case FG_OUTPUT_SET:
    FG->output = (u_int)arg;
#endif
    ifdef PCMCIA_DEBUG
    if (pc_debug)
        printk(" FG_OUTPUT_SET %hi\n", (u_int)arg);
    #endif
    break;
case FG_STD_SET:
    if ((u_char)arg == STD_MTSC) {
        FG->std = STD_MTSC;
        FG->width = MTSC_WIDTH;
        FG->height = MTSC_HEIGHT;
        VDECODER = VDECODER_MTSC;
        for (x=0; x<0x1D;x++) {
            cardcam_SetVD(x,VDECODER[x]);
        }
    } else if ((u_char)arg == STD_PAL) {
        FG->std = STD_PAL;
        FG->width = PAL_WIDTH;
        FG->height = PAL_HEIGHT;
        VDECODER = VDECODER_PAL;
        for (x=0; x<0x1D;x++) {
            cardcam_SetVD(x,VDECODER[x]);
        }
    } else {
        rtn = -EINVAL;
    }
    break;
case FG_STD_GET:
    put_user_byte(FG->std, (u_char *) arg);
    break;
default:
    return -ENOMEM;
    return rtn;
}
static void fg_close(struct inode *inode, struct file *file)
{
    #ifdef PCMCIA_DEBUG
    if (pc_debug)
        printk("cardcam_cs: fg_close()\n")
    #endif

}
static void cardcam_SetVD(int reg,int val)
{
    outw(reg<<8,VDA);
    outw(val<<8,VDC);
}
static char cardcam_GetVD(int reg)
{
    outw(reg<<8,VDA);
    return ((inw(VDC)>>8) & 0xff);
}

B.3 The fgraber Startup Script File

#!/bin/sh
#
# $Id: fgraber,v 1.2 1996/05/24 21:10:52 pchan Exp $
#
# Initialize or shutdown a PCMCIA video capture card

124
 Modify from David Hinds' pcmem script

The first argument should be either 'start' or 'stop'. The second argument is the base name for the device. When starting a device, there should be two additional arguments, the major and minor device numbers.

usage()
{
    echo "usage: fgrabber [action] [device name] [major] [minor]"
    echo " actions: start check stop suspend resume"
    exit 1
}

if [ $# -lt 2 ] ; then usage ; fi

action=$1
name=$2

case "$1" in
    'start')
        if [ $# -ne 4 ] ; then usage ; fi
        major=$3
        minor=$4
        rm -f /dev/$name
        mknod /dev/$name c $major $minor
        ;;
    'check')
        fuser -s /dev/$name &> exit 1
        ;;
    'stop')
        fuser -s -k /dev/$name
        if mount | grep "-/dev/$name" on" ; then
            umount /dev/$name || exit 1
        fi
        rm -f /dev/$name
        ;;
    'suspend'|'resume')
        ;;
    *)
        usage
        ;;
esac

exit 0
Bibliography


