

Copyright © 1996, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**PEN AND SPEECH RECOGNITION IN THE USER
INTERFACE FOR MOBILE MULTIMEDIA TERMINALS**

by

Shankar Narayanaswamy

Memorandum No. UCB/ERL M96/11

25 March 1996

**PEN AND SPEECH RECOGNITION IN THE USER
INTERFACE FOR MOBILE MULTIMEDIA TERMINALS**

Copyright © 1996

by

Shankar Narayanaswamy

Memorandum No. UCB/ERL M96/11

25 March 1996

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Pen and Speech Recognition in the User Interface for Mobile Multimedia Terminals

by

Shankar Narayanaswamy

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Robert W. Brodersen, Chair

Portable computers have soared in popularity over the last few years. Vendors are introducing new models with smaller form factors, longer battery life, communications capabilities and unique user interfaces using pen or audio input.

The design and implementation of a networked user interface architecture using handwriting recognition and speech recognition is explored. Although the user interface was designed for mobile multimedia terminals such as the InfoPad system, it is more generally applicable in any application domain where pen and/or spoken input are preferable to keyboard input.

We examine the kinds of handwriting and speech recognizers needed to provide an effective user interface. There are several aspects to this problem. Firstly, there is the user interaction model which determines where and how the user uses each input modality. Secondly, there is the applications programming model which determines the level of abstraction and the extent of encapsulation of the recognizer's functionality. Thirdly, there is the service provision model which determines whether the recognizer is part of the application or whether it runs as a separate thread or process, or somewhere in between. The latter allows off-loading the recognition computation onto a remote, possibly specialized server and minimizes the impact of compute-intensive recognizers on other applications.

The entire infrastructure for a pen and speech based user interface is described, including a software hidden Markov model based writer-independent hand-print recognizer, a VLSI hidden Markov model based large-vocabulary speaker independent continuous speech recognizer, type servers to handle the new data types and applications that exercise the entire user interface architecture.



Professor Robert W. Brodersen
Committee Chairman

Table of Contents

Acknowledgments.....	xii
CHAPTER 1. Introduction.....	1
1.1. Motivation	2
1.1.1. Portable Computers/Communicators	2
1.1.1.1. Size Constraint.....	2
1.1.1.2. Weight Constraint.....	3
1.1.1.3. Other Considerations	3
1.1.2. Computer Aided Design	4
1.1.3. Persons With Disabilities	4
1.2. Enabling Technology	5
1.3. Pen Digitizer Technology.....	5
1.4. Limitations of Pen/Speech Interfaces.....	6
1.4.1. Pen Digitizers.....	7
1.4.2. Speech Recognizers and Noise	7
1.4.3. Recognition Errors	8
1.5. Previous Work.....	8
1.5.1. Pen Interfaces.....	8
1.5.2. Speech Interfaces	10
1.5.3. Pen and Speech Interfaces	11
1.6. Scope of Thesis	11
CHAPTER 2. The InfoPad System.....	12
2.1. System Design.....	12
2.1.1. InfoPad Applications	13
2.1.2. Design Objectives and Constraints	15
2.1.3. Design Choices	15
2.2. Wired Network Architecture.....	16
2.3. Terminal Design	17
2.4. Pen Subsystem on IPGraphics.....	21
2.5. Pen and Keyboard Subsystems on IPVideo	22
CHAPTER 3. The User Interface Architecture	28
3.1. Need for a New Architecture.....	28
3.2. High Level Description	29
3.3. Pen Server.....	32
3.3.1. Mouse Emulation	32
3.3.2. Pen-Resolution Data	33
3.3.3. General Operation	34
3.3.4. Command Line Options	36
3.3.5. Applications Programming Interface	36
3.3.6. Pen Support for X Workstations	37

3.4.	Audio Server.....	38
3.4.1.	AudioFile Compatibility	38
3.4.2.	Enhancements Over AudioFile	39
3.5.	Handwriting and Speech Recognizers.....	40
3.5.1.	Handwriting Recognizer	40
3.5.2.	Speech Recognizer.....	41
3.5.3.	Servers Versus Software Libraries.....	42
CHAPTER 4. Handwriting Recognition		43
4.1.	Previous Work.....	43
4.1.1.	GO Corporation	43
4.1.2.	IBM	44
4.1.3.	AT&T	44
4.1.4.	CIC	45
4.1.5.	Paragraph	45
4.1.6.	Apple.....	45
4.2.	Characteristics of Handwriting Recognizers.....	46
4.2.1.	Vocabulary	46
4.2.2.	Grammar	46
4.2.3.	Constrained Writing Area.....	46
4.2.4.	Spatial Locality	47
4.2.5.	Digitizer Resolution	47
4.2.6.	Available Computational Power	48
4.3.	Recognition Requirements of the InfoPad	48
4.3.1.	Non-Dictionary Words.....	48
4.3.2.	Mass Text Entry.....	48
4.3.3.	Gestures.....	49
4.3.4.	Geometric Shapes	49
4.4.	Models for Providing Recognition Services	49
4.4.1.	User Interaction Model	49
4.4.2.	Programming Model	50
4.4.2.1.	Uncoupled Applications	50
4.4.2.2.	Loosely Coupled Applications	51
4.4.2.3.	Tightly Coupled Applications	51
4.4.3.	Service Provision Model.....	51
4.5.	Properties of Printed Handwriting.....	52
4.6.	HMM Based Handwriting Recognition	53
4.6.1.	Hidden Markov Modeling.....	54
4.7.	A Writer Independent Handprint Recognizer	57
4.7.1.	The Recognition Algorithm	57
4.7.1.1.	Heuristics	57
4.7.1.2.	Preprocessing and Feature Extraction	57
4.7.1.3.	The Hidden Markov Model	58
4.7.1.4.	Viterbi Algorithm	61
4.7.1.5.	Post-Processing.....	62

4.7.2.	Character Sets	62
4.7.3.	Applications Programming Interfaces	62
4.7.3.1.	Handwriting Recognition Widget.....	63
4.7.3.2.	Remote Server	64
4.7.3.3.	Sun API	65
4.7.4.	Data Capture and Manipulation	67
4.7.5.	Training Set.....	73
4.7.6.	Performance	75
4.7.7.	Algorithmic and Implementational Improvements	76
CHAPTER 5.	Speech Recognition	78
5.1.	Performance of Existing Systems.....	78
5.1.1.	Software	79
5.1.2.	Off-the-Shelf Hardware	79
5.1.3.	Custom Hardware	79
5.1.4.	Accuracy on Realistic Tasks	79
5.2.	Characteristics of Speech Recognizers.....	80
5.2.1.	Word Length	80
5.2.2.	Vocabulary Size	81
5.2.3.	Grammar	81
5.2.4.	Available Computational Power	82
5.3.	Recognition Requirements of the InfoPad	82
5.3.1.	Commands	82
5.3.2.	Dictation.....	83
5.4.	Models for Delivering Recognized Speech.....	84
5.4.1.	User Interaction Model	84
5.4.1.1.	Speech Recognition Focus.....	84
5.4.2.	Programming Model	85
5.4.2.1.	Uncoupled Applications	85
5.4.2.2.	Loosely Coupled Applications	85
5.4.2.3.	Tightly Coupled Applications	86
5.4.3.	Service Provision Model.....	86
5.5.	A Small, Flexible Recognizer	87
5.5.1.	Motivation.....	87
5.5.2.	Implementation	88
5.5.3.	Discussion	90
5.6.	A Real Time Large Vocabulary Speaker Independent Speech Recognizer	91
5.6.1.	Differences from the Small, Flexible Recognizer.....	92
5.6.2.	The Recognition Algorithm	92
5.6.2.1.	Feature Extraction.....	93
5.6.2.2.	Hidden Markov Modeling	94
5.6.2.3.	Viterbi Algorithm	95
5.6.2.4.	Backtrace	98
5.6.3.	System Architecture.....	98
5.6.3.1.	Changes to Improve Performance	98

5.6.3.2.	Hierarchy	99
5.6.3.3.	System Hardware Partitioning.....	101
5.6.4.	Phone Process	102
5.6.5.	Active Word Process.....	102
5.6.5.1.	Description.....	103
5.6.5.2.	Implementation.....	105
5.6.5.3.	Request Processor.....	107
5.6.5.4.	Probability Processor.....	108
5.6.5.5.	Grammar Node Processor.....	109
5.6.6.	Viterbi Board Design	109
5.6.7.	System Level Simulation	111
CHAPTER 6.	Applications.....	114
6.1.	Circuit Schematic Recognizer	114
6.1.1.	Desired Functionality.....	114
6.1.2.	Current Systems	115
6.1.3.	Lessons Learned.....	116
6.1.4.	Implementation	118
6.1.5.	File Format.....	122
6.1.6.	Suggested Improvements.....	124
6.2.	Electronic Notebook.....	125
6.2.1.	Desired Functionality.....	126
6.2.2.	Lessons Learned.....	126
6.2.3.	Implementation	127
6.2.4.	Suggested Improvements	129
CHAPTER 7.	Conclusions.....	130
7.1.	Summary of Results	130
7.1.1.	User Interface Architecture.....	130
7.1.2.	Handwriting Recognition.....	131
7.1.3.	Speech Recognition	132
7.1.4.	Applications	132
7.2.	Future Work	132
7.2.1.	Development Projects	133
7.2.1.1.	Audio Focus Manager	133
7.2.1.2.	Pen Server Control Widget.....	133
7.2.1.3.	Handwriting Recognition Widget.....	134
7.2.1.4.	Speech Recognition Widget	134
7.2.1.5.	User Interface Control Widget	134
7.2.1.6.	Other Improvements	135
7.2.2.	Research Projects	135
7.2.2.1.	Complex Event Manager	135
7.2.2.2.	Synergy Between Handwriting and Speech Recognition.....	136
7.2.2.3.	Handwriting and Speech Recognition	136
7.2.2.4.	Integrated Document Editor	137

CHAPTER 8. Appendix: Handwritten Character Sets	138
CHAPTER 9. Appendix: Software Organization	145
9.1. Pen Server.....	145
9.2. Audio Server.....	145
9.3. Handwriting Recognizer.....	145
9.4. Circuit Schematic Recognizer	145
9.5. Notebook Application	146
Bibliography.....	154

List of Figures

CHAPTER 1.	1
Figure 1-1. Buttons on a Typical Pen and a Typical Mouse.....	7
CHAPTER 2.	12
Figure 2-1. Diagrammatic Representation of the InfoPad System.....	14
Figure 2-2. The InfoPad Network (InfoNet) Architecture	17
Figure 2-3. InfoPad Type Servers.....	18
Figure 2-4. Architecture of the IPGraphics Terminal.....	18
Figure 2-5. Architecture of the IPVideo Terminal	19
Figure 2-6. IPGraphics Pen Subsystem.....	21
Figure 2-7. Protocol Chip Photograph	23
Figure 2-8. IPVideo Pen and Keyboard Subsystems.....	24
Figure 2-9. BusMaster Module for IPVideo Pen and Keyboard Subsystem....	25
Figure 2-10. Schematic of UART Used in Pen and Keyboard Module.....	26
Figure 2-11. PenAudio Chip Photograph.....	27
CHAPTER 3.	28
Figure 3-1. InfoPad User Interface Architecture.....	30
Figure 3-2. Pen Server Packet Structure.....	34
CHAPTER 4.	43
Figure 4-1. High-Level Description of a HMM Based Handwriting Recognizer	54
Figure 4-2. Topology of an 8-State Handwriting Markov Model	60
Figure 4-3. Screen Shot of the Handwriting Recognition Widget	63
Figure 4-4. Main Menu Bar for the Data Capture and Manipulation Package ..	67
Figure 4-5. Example of Word File Format	68
Figure 4-6. Feature File Syntax	69
Figure 4-7. Vector File Syntax	70
Figure 4-8. Handwriting Capture Canvas.....	70
Figure 4-9. Global Variable Display/Editing Window	72
Figure 4-10. HMM Parameter File Syntax.....	74
CHAPTER 5.	78
Figure 5-1. GramCracker Application for Creating and Modifying Speech Recognition Vocabularies	89
Figure 5-2. Code Fragment to Illustrate API for Speech Recognizer	90
Figure 5-3. Block Diagram of the Feature Extraction Algorithm	93
Figure 5-4. Graphical Representation of a Hidden Markov Model.....	94

Figure 5-5.	Trellis for Decoding a Hidden Markov Model	96
Figure 5-6.	Source and Destination Grammar Nodes	100
Figure 5-7.	Hardware Partitioning of the Speech Recognition System	101
Figure 5-8.	Connections to the Active Word Process.....	103
Figure 5-9.	Implementation of the Active Word Process	105
Figure 5-10.	Chip Photograph of the Request Processor	107
Figure 5-11.	Chip Photograph of the Probability Processor	108
Figure 5-12.	Chip Photograph of the Grammar Node Processor	109
Figure 5-13.	Switching Architecture of the Viterbi Board.....	110
CHAPTER 6.	114
Figure 6-1.	Screen Shot of the Circuit Schematic Recognizer with Edit Menu Posted	118
Figure 6-2.	Parameter Editing Form for MOS Transistors.....	120
Figure 6-3.	Speech Recognition Control Widget	121
Figure 6-4.	Circuit Schematic File Example	123
Figure 6-5.	Circuit Schematic File Syntax	123
Figure 6-6.	Screen Shot of Electronic Notebook Application.....	127
Figure 6-7.	Electronic Ink Stroke and Word Structures	128
Figure 6-8.	Electronic Ink File Format.....	129
CHAPTER 7.	130
CHAPTER 8.	138
CHAPTER 9.	145

List of Tables

CHAPTER 1.	1
Table 1-1. Some PDAs Currently on the Market.....	1
Table 1-2. Characteristics of Current Pen Digitizers	6
CHAPTER 2.	12
Table 2-1. Signal Pins on the Pen Module.....	22
CHAPTER 3.	28
Table 3-1. Pen Packet Structure for the Logitech Gazelle Digitizer.....	33
CHAPTER 4.	43
Table 4-1. Heuristics Used in Handwriting Recognition.....	57
Table 4-2. Number of Markov States for Each Handwritten Character in the 61-Character Recognizer	58
Table 4-3. Number of Markov States for Each Handwritten Character in the Digit Recognizer.....	60
Table 4-4. Handwriting Recognition Results.....	76
CHAPTER 5.	78
Table 5-1. Accuracies for Recent Speech Recognizers	80
Table 5-2. Contents of the Active Word List	103
Table 5-3. Memory Sizes on the Viterbi Board	111
Table 5-4. Components on the Viterbi Board	112
CHAPTER 6.	114
Table 6-1. List of Speakable Commands	121
CHAPTER 7.	130
CHAPTER 8.	138
Table 8-1. Character Set for 61-Character Recognizer.....	139
Table 8-2. Character Set for Digit Recognizer	144
CHAPTER 9.	145
Table 9-1. Pen Server Source Tree	146
Table 9-2. Audio Server Source Tree	147
Table 9-3. Handwriting Recognizer Source Tree	148
Table 9-4. Data Capture and Manipulation Package Source Tree	149
Table 9-5. Handwritten Data Source Tree	150

Table 9-6.	Hidden Markov Model Parameters Source Tree	151
Table 9-7.	Circuit Schematic Recognizer Source Tree.....	152
Table 9-8.	Notebook Source Tree	153

Acknowledgments

Many people deserve my thanks and appreciation for their contribution to the successful completion of my graduate school career. My family was always there for me with unfailing emotional, spiritual, and material support. They never let me lose sight of my goal of getting a Ph. D. and encouraged me onwards even when they were not entirely convinced that I was doing the right thing.

Professor Robert Brodersen, my research advisor, was an inspiration with his leadership and vision. From him I learned to not get bogged down in details but instead to always look at the big picture and constantly re-evaluate the value of my work. Professor Jan Rabaey was always available for technical and other advice. I am grateful for his cheerful, thoughtful input.

Working closely with Anton Stolzle during the first 3 years of graduate school was a pleasure. He was always available to discuss issues in speech recognition or hardware design, or just about anything else. I had the pleasure of interacting with Brian Richards on numerous occasions. He always responded quickly to Lager problems and was a wellspring of knowledge about both hardware and software design. He also designed early versions of the Pen Server and the UART used in the IPVideo chip.

Andrew Burstein's hidden Markov model speech recognition code was the starting point for my handwriting recognizer. I appreciate his responsiveness to questions about his code, hidden Markov modeling and other software questions, especially while I was learning to program in C++.

Jeff Gilbert made major contributions to the Audio Server, without which its performance would not have been as good. His cheerful can-do attitude even in the face of tremendous demo deadline pressure made my interactions with him enjoyable, even when he was the group leader!

I would also like to acknowledge Armando Fox for implementing the handwriting widget interface and Steve Chow for doing a large part of the handwriting data capture, segmentation and labeling.

1 Introduction

Portable computers have soared in popularity in the last few years. Several vendors have released Personal Digital Assistants (PDAs) which have a small form factor and long battery life (see Table 1-1) [Pen 95]. Many PDAs use pen interfaces rather than keyboards and

Vendor	Model	Screen Size (inches)	Weight (pounds)	Input modes	Handwriting Recognition?	Battery Life (hours)
Apple	MessagePad 120	3.8 x 2.8	1.28	pen	Yes	22
Sharp	Zaurus	4.0 x 2.6	0.85	pen + keyboard	No	60
Casio	Z-7000	3.1 x 4	0.95	pen	Yes	90
Amstrad	PIC 700	3.4 x 5.4	0.875	pen	Yes	40
Sony	PIC-1000	3.0 x 4.5	1.2	pen +optional keyboard	Yes (3rd party)	12

Table 1-1. Some PDAs Currently on the Market

mice. A pen digitizer is smaller and lighter than a keyboard and provides a more natural user interface, but requires a much more sophisticated user interface infrastructure. Some experts predict that within a few years we will see wide availability of small personal computers with pen and voice input [Cran93].

The main feature of portable computers that limits their ability to provide a sophisticated user interface is their limited computational capacity. However recent advances in technology have made it possible to use more sophisticated user interfaces in portable terminals and computers, and this thesis examines the issue of extending the user interface to use both handwriting and speech as input modes. Such a user interface is very useful for portable computers and is also applicable to desktop computers in situations where a pen and speech interface would be advantageous.

1.1. Motivation

There are three major scenarios where a pen/speech interface is preferred: portable computers/communicators, Computer Aided Design (CAD) platforms, and computing for persons with disabilities. In general, it is worth noting that handwriting and speaking skills are acquired early in life. They perform an important role in human-human communication, and are therefore better suited to human-computer communication from the human's point of view [Rhyn93].

1.1.1. Portable Computers/Communicators

Portable computers are designed with strict attention to size and weight constraints. As explained below, the pen is clearly superior to a keyboard and mouse in both size and weight. If a pen digitizer is used in place of a keyboard and mouse, we can overcome the disadvantages of the latter devices. The pen can easily replace the mouse as a pointing device and, with handwriting recognition, the pen can also replace the keyboard in many situations. In order to use the pen and speech as an alternative to the keyboard, we need to create a sophisticated user interface using handwriting and/or speech recognition that is comfortable for users. We also need a good applications programming interface to encapsulate and abstract the recognizers.

1.1.1.1. Size Constraint

The size of a portable computer or communicator affects its user acceptability. A device that fits into a pocket is very convenient while a terminal that is too large to fit into a briefcase is not convenient. A keyboard adds significant volume to the size of a portable computer. For example, a regular PC-style keyboard occupies 18.5 by 7 by 1.5 inches. The keyboard on an IBM Thinkpad 755C occupies 11.5 by 6 by 0.5 inches, while the keyboard on a Hewlett Packard HP200LX palmtop computer occupies 6 inches by 2.5 inches by 0.6 inches. When in use, these keyboards occupy valuable space and make the devices more cumbersome, thereby making them less usable. The smaller keyboards are also difficult to use since it is not easy to fit all ten fingers into the limited dimensions of a small keyboard.

Many recent portable computers use a TrackPoint mouse (a little stub that sticks up from the middle of the keyboard) to provide pointer input. This eliminates the need for a cumbersome external mouse. However, this mouse is not as easy to use as a normal mouse.

The size-related cost of using a pen digitizer is a thicker screen incorporating the digitizer and storage space for the pen. For speech, the size related cost is an audio codec and a microphone jack.

1.1.1.2. Weight Constraint

A regular IBM PC keyboard weighs between 2 and 4 pounds and a Microsoft mouse weighs 3.5 ounces (6 ounces with cable), whereas a Logitech Gazelle PenMan digitizer with pen weighs about 3.5 ounces. The weight savings resulting from using a pen digitizer rather than a keyboard and mouse are considerable, especially considering that the target weight for modern portable computers is below 4 pounds.

1.1.1.3. Other Considerations

A keyboard-based computer requires two hands to operate and a stable surface upon which to place the computer. A pen interface requires only one hand to operate; the other hand is free to do other things or to support the computer.

The pen is arguably a better pointing device than a mouse. It is advantageous to use the pen on the screen at the exact position where the user wants to activate a menu. However, the disadvantage is that the user has to move his entire arm to use the pen whereas with a mouse the total distance of arm travel is lower. This is a result of the relative coordinate system and the acceleration/magnification provided by the mouse. For small screens this problem is not severe.

A parallax problem arises when a pen digitizer with an LCD screen overlay is used. The thickness of the LCD screen's glass causes a separation of between 1 and 2 mm between the pen tip and the LCD display. The user's perception of pen position is therefore not the same as the device's knowledge of pen position. Additionally, there is some variation in the calibration and registration of pen digitizers from the same vendor so digitizer param-

eters may vary from sample to sample. However, this problem may be partially alleviated by using software alignment/registration.

Entering text via typing is at least twice as fast as hand printing. Touch typists can achieve a rate of 6-7 characters/sec while printing can be done at 1-1.3 characters/sec. Unistrokes, which are faster to write than handprinted characters, can be written at 3.4 characters/sec. [Gold93] [Tapp90]. Using a pen for mass text input such as writing a book or programming is not as efficient as using a keyboard. Therefore a keyboard is preferred over a pen in applications where mass text entry is required.

1.1.2. Computer Aided Design

Computer Aided Design (CAD) tools are used in many industries. Design often involves freehand drawing (as in the garment design industry) or creating designs which comprise only a few primitives (such as in digital and analog integrated circuit design). These applications can be done more efficiently using a pen than a mouse and keyboard. For example, in the garment design industry it is easier to draw a design than to specify a spline for each curve using a mouse. In circuit design, it is more convenient to draw each circuit element than to type its name or select it from a menu. Traditional circuit design systems require the user to navigate using a mouse and to enter information on a keyboard, which requires constant switching between the two devices. This issue is examined in greater detail in Section 6.1. on page 114.

1.1.3. Persons With Disabilities

Persons suffering from tendonitis or carpal tunnel syndrome are unable to use a keyboard for any great length of time. However, they can usually use a pen for much more time, and can speak all day. A pen/speech interface would be ideal for such persons. In fact, many disabled persons currently use commercial voice recognition products such as DragonDictate and Kurzweil Voice for dictation and even programming [Cros95]. A Wacom or other digitizer provides a pointing device facility in cases where screen navigation cannot be done via text commands.

1.2. Enabling Technology

Currently, a major factor preventing the use of a sophisticated recognition-based user interface is the high computational cost of providing recognition services at adequate performance levels. Performance measures include real-time operation and high accuracy. Advances in low power, high speed microprocessors and in communications have now made it possible to incorporate such a user interface in a cheap, lightweight terminal.

Computer technology has been advancing very quickly in the last decade. We now have extremely powerful microprocessors in desktop and portable computers. Processors are cheap; they can be embedded in application specific hardware for performing specialized tasks and, for a given desktop or portable, there are usually several embedded processors performing specialized functions such as graphics rendering, sound processing, and communications. Advances in low power circuit design make it possible to use embedded processors in portable devices. It is therefore possible to embed specialized recognition or compression/decompression hardware into portable devices to provide the required computational power.

Recent advances in networks and in wireless communications make it possible to off-load computation onto a remote compute server rather than run all user and recognition processes on the local processor. This allows access to greater computational resources than can be provided on an isolated mobile terminal. The InfoPad system described in Chapter 2 uses this approach to enable a powerful user interface on a dumb terminal.

1.3. Pen Digitizer Technology

There are currently several vendors of pen digitizers (see Table 1-2).

The Logitech Gazelle digitizer uses electromagnetic technology. In this technology, a battery-powered wireless pen transmits a signal containing information on switch status and battery level. A digitizer circuit board containing traces and other circuitry detects the pen signal and computes the pen coordinates. The digitizer board is mounted below the LCD screen.

Model	Size (inches)	Resolution (lines/inch)	Sample Rate (points/s)	Accuracy (inches)	Technology
Logitech Gazelle PenMan	8.1x5.85	414	377	0.01	electromagnetic
Wacom SD-510C	9.13x5.91	500	100	0.02	electromagnetic resonance
Wacom UD-0608-R	8x6	1270	205	0.01	electromagnetic resonance
Scriptel	12x12	1000	200	0.015	resistive decoding

Table 1-2. Characteristics of Current Pen Digitizers

Wacom digitizers use electromagnetic resonance technology where the tablet transmits a signal to a batteryless and cordless pen for 20 μ sec, then receives a signal which is re-emitted by the pen at a different frequency for 20 μ sec. Tablet coordinates are computed based on signal strength across several grid wires under the tablet surface.

The Scriptel digitizer uses resistive decoding technology. A wired pen transmits a signal encoding button status information which is detected by a resistive film deposited on transparent glass. Pen coordinates are calculated based on voltages measured at the edges of the resistive film. Since the digitizer is transparent, it may be mounted above the LCD screen, replacing the glass that usually protects the screen from mechanical damage.

There are other pen digitizers which use purely resistive technology. In this case, direct pressure of the pen (or any other object, such as a finger) on the digitizer closes microswitches or modifies local resistivity. The change is detected and used to compute pen coordinates. The disadvantage of this technology is that there is no information about pen trajectory when it is not on the digitizer. Raised strokes therefore cannot be detected.

1.4. Limitations of Pen/Speech Interfaces

There are several limitations inherent in pen/speech interfaces. In the following subsections, we examine each of these limitations and discuss some ways to overcome them.

1.4.1. Pen Digitizers

Although the pen is an excellent pointing device, it requires significant movement of the user's hand across the screen. A mouse amplifies the user's hand movements and reflects these movements on the screen. The pen also has fewer buttons than a typical modern mouse (see Figure 1-1). A mouse on a workstation or PC usually has three buttons which

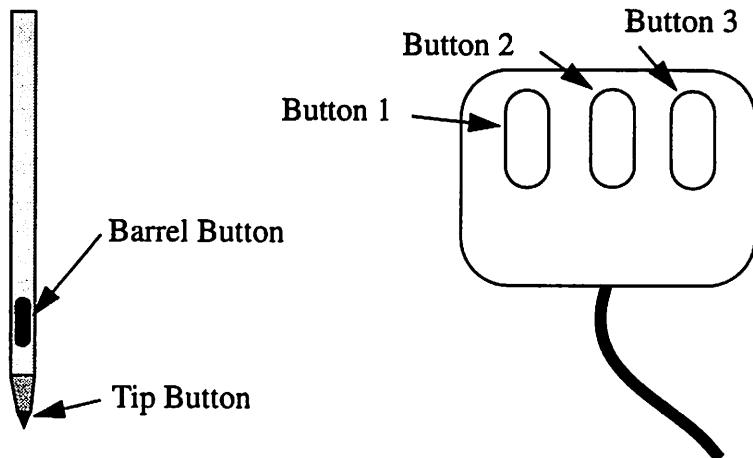


Figure 1-1. Buttons on a Typical Pen and a Typical Mouse

are easily distinguishable from one another, whereas a digitizing pen has one button on the tip and sometimes another on the barrel. The barrel button is difficult to use because it is positioned such that users often press it by accident. When called upon to press the barrel button, users often have difficulty locating the button since its position is not intuitive. Therefore, the pen has effectively only one input button, and this button is activated by bringing the pen into contact with the screen. This limitation on the number of input buttons can be alleviated by placing additional buttons or modifying keys on the computer or terminal itself. In the case of modifying keys, the user would press the modifier key and then tap the pen on the screen to emulate additional pen buttons.

1.4.2. Speech Recognizers and Noise

Current speech recognizers do not perform well in noisy environments. Interfaces that use speech recognition are therefore not very tolerant of noise yet they generate noise. A roomful of people speaking to their terminals may create so much noise that not only will everyone get annoyed, the recognizers may fail due to a low signal-to-noise ratio. A speech recognizer will also not perform well in a naturally noisy environment such as a factory

floor or a meeting room. However, using headphones and an echo-cancelling microphone can help alleviate these problems.

1.4.3. Recognition Errors

Handwriting and speech recognizers both make frequent recognition errors. Chapter 4 and Chapter 5 report error rates of current handwriting and speech recognition systems respectively, but these numbers are obtained from artificial tasks in laboratory conditions. Under realistic conditions the error rates are much higher.

Error rate increases with larger vocabularies and less constrained grammars. Error rates are greater for writer (or speaker) independent systems than for dependent systems which are tailored for a particular user. Cursive handwriting recognizers tend to have greater error rates than print recognizers; similarly continuous speech recognition systems tend to make more errors than isolated-word speech recognition systems. We can reduce the effects of a finite recognition error rate by reducing the vocabulary, using a more constraining grammar and by providing error correction facilities.

1.5. Previous Work

There has been some research on pen interfaces and speech interfaces, but little work has been done on using both pen and speech together. This section reviews some previous work in these three areas.

1.5.1. Pen Interfaces

As shown in Table 1-1 on Page 1, there are several PDAs with pen-based interfaces currently on the market. The Apple MessagePad uses the Newton operating system. The other PDAs use either Windows for Pen by Microsoft, OS/2 for Pen by IBM, PenDOS by CIC, or GEOS by Geoworks. There are no commercial pen interfaces based on UNIX.

Goldberg and Goodisman [Gold91] at Xerox PARC examined how to interface to handwriting recognition algorithms and the effective exploitation of the differences between a pen and a keyboard/mouse. This was in the context of the design of a text entry tool. They chose to use print rather than cursive input since there were no cursive systems known to the authors which had consistently high recognition rates. The system recognized both

upper and lower case print. They chose to do all their user interface studies in boxed mode because recognition in this mode is more accurate. Correction and editing are also simpler. The greatest advantage of boxed mode entry is that users do not slip into script mode and join adjacent letters.

The system recognizes single letters rather than words and provides two correction gestures for deletion and insertion. The most interesting trade-off is the choice of writer dependence versus independence. They chose to use a writer dependent system due to its better accuracy. Training is explicit but without a separate training mode. The user can modify the training samples whenever the recognizer makes an error during normal operation.

Rhyne, Chow and Sacks [Rhyn91] at the IBM T. J. Watson Research Laboratory worked on adding a paper-like interface (PLI) and handwriting recognition to the X Window System. The X Window System was enhanced to provide a stylus-based user interface for handheld computers. Modifications were done to the X server itself and to the Xt toolkit. A new widget called a WritingArea widget was created to receive strokes from the X server and which invokes application-supplied callback functions. This stroke-receiver widget maintains a list of the active input strokes. Another widget called a WritingReco widget provides recognition services on top of the services supplied by the WritingArea widget. These additional services include error correction, prototype management and recognizer management functions.

Goldberg and Richardson [Goldberg93] at Xerox PARC designed an approach to stylus touch-typing using an alphabet of unistrokes, which are letters specially designed to be used with a stylus. Unistrokes are faster to write, less prone to recognition errors, and require very little screen real estate. They also can also be entered in an “eyes-free” manner because they consist of single strokes.

The set of unistrokes were chosen to be robust in the face of sloppy writing. In the authors’ system, letters could be written one on top of the other, so the screen real estate used is very small. Many unistroke characters are similar to Roman letters. The stylus’ barrel button is used to denote capitals, thereby reducing the size of the alphabet of unistrokes by a factor

of two. Writing time for each unistroke varied from 150 to 300 msec. The median inter-stroke time was 158 msec.

Kurtenbach and Buxton [Kurt94] at the University of Toronto worked on marking menus, which allow a user to perform menu selection either by popping up a radial menu or by making a straight mark in the direction of the desired menu item without popping up the menu. They found that when users become expert with the menus, marks are used extensively. Also, using a mark was on average 3.5 times faster than using the menu.

The Windows for Pen extension to the Microsoft Windows operating system provides very similar services [Micr95]. Windows for Pen also provides pixel level callbacks where the PLI system provides only stroke level callbacks, and has an extensive library of routines for handling electronic ink and recognized handwriting.

The main difference in approach is that PLI tries to encapsulate as much of the pen functionality as possible whereas Windows for Pen allows applications programmers access to all details of the process of handwriting capture.

1.5.2. Speech Interfaces

It is possible to use speech to navigate between windows on the screen. The Xspeak system from the Massachusetts Institute of Technology [Schm90] is an application running on X workstations that recognizes spoken window names for navigation purposes. Xspeak partially replaces the mouse for navigation and for issuing commands to the window manager. Experiments showed that speech recognition can create a viable user interface for navigation among windows.

The greatest limitation to Xspeak's functionality is the need to associate a name with each window. Several windows can have the same name, especially in the X window system, and this can confuse Xspeak. Also, window names may not be dictionary words and so need to be trained separately. In Xspeak, a control panel allows the user to adaptively specify the spoken name for each window.

1.5.3. Pen and Speech Interfaces

Rhyne and Wolf [Rhyn93] at the IBM T. J. Watson Research Laboratory identified and examined issues relating to recognition-based interfaces. They found that recognition feedback, correction and training are critical to the success of any recognition-based interface. Recognition feedback allows the user to detect that an error has occurred. Correction mechanisms are needed to allow the user to easily correct recognition errors. User training can be less frustrating and more efficient for the user if it is done at once rather than in the context of use, although it may not be as effective.

1.6. Scope of Thesis

Chapter 2 describes the InfoPad system [Chan93], which is the primary target system for this user interface architecture. The InfoPad system is designed around a prototype mobile multimedia terminal built here at the University of California at Berkeley. It has digitized pen and speech input rather than a keyboard and mouse. Chapter 3 describes the user interface architecture that I designed and implemented to take advantage of handwritten and spoken input in the system and to provide services related to this input paradigm. This architecture uses a networked client-server model to distribute the computation of the components of the user interface and to provide services; it is equally applicable to InfoPad as well as to other environments.

Chapter 4 describes a handwriting recognition engine which uses hidden Markov models and very simple feature vectors to recognize printed characters. Chapter 5 describes the InfoPad system's speech recognition requirements and describes the design and implementation of a large vocabulary speaker independent connected speech recognizer, which required 6 custom VLSI chips [Stol92]. This recognizer could be used as a speech recognizer server for the InfoPad system. Chapter 6 describes applications that were built to prove that pen and speech form a viable user interface for some class of systems and applications. The primary application is a circuit schematic recognizer that recognizes drawn circuit elements and wires. It uses the entire user interface architecture, including raw pen data, recognized handwriting, and recognized speech. Chapter 7 draws conclusions and describes directions for future work.

2 The InfoPad System

The User Interface Architecture described in this thesis was designed and implemented in the context of the InfoPad system [Chan93]. The InfoPad project is a multi-disciplinary research effort to build a personal communications system for ubiquitous computing. Research areas include low-power integrated circuit and terminal design, high-bandwidth radio transceiver design, high-speed multimedia networking, integrated mechanical and electrical terminal design, and advanced user interface design. Several faculty members and many students at Berkeley are involved in building the system and examining the relevant issues.

This chapter describes the InfoPad system, concentrating on the elements that affect the User Interface Architecture. The system, network, and terminal design are described first. The Pen Subsystem for two generations of the InfoPad terminal is then described.

2.1. System Design

The design of the InfoPad system is a research effort that explores one extreme of the design space for mobile personal communications systems. It includes a dumb terminal with a high-bandwidth connection to a high-speed network, as described below. There are two basic assumptions that drive the system design. The first assumption is that the wired backbone network is fast enough to provide a very high bandwidth with low latency. The second is that given the fast network, a collection of network-connected compute servers is available on which to run user and system processes. These two assumptions are based on the premises that network bandwidth is available and that computation is also available and best done in the network.

A diagrammatic representation of the system is shown in Figure 2-1. Each terminal has a high-bandwidth connection to a basestation that is on a high-speed wired network. The network also supports compute servers and sources for multimedia information, such as video databases. The applications that are supported and the design objectives and constraints faced are described in this section.

2.1.1. InfoPad Applications

The InfoPad system is designed for mobile multimedia information retrieval and for mobile personal communications. Examples of applications requiring multimedia information retrieval may be found in libraries, museums, and repair depots. In a library, users currently use a fixed terminal to query a database for a call number which is used to locate the book. However with a mobile, multimedia-enabled terminal they can view a map of the library showing the location of the book. If a user is searching for a video, he could view it on the terminal rather than going to a special viewing room.

In a museum, the user may retrieve information about the painting or sculpture he is currently viewing, as well as information about the artist and his work. In a repair depot, a mechanic may pull up schematics or other documentation on the part he is currently servicing. Of course, the most general current application requiring multimedia information retrieval and display is surfing the World Wide Web.

In personal communications, the user may want to send and retrieve multimedia electronic mail, make a telephone call, or videoconference. We support 1-way videoconferencing only (1-way video broadcast with 2-way audio) since there is no camera on the terminal. Future versions of the InfoPad terminal may incorporate a small, light camera and thereby support true 2-way videoconferencing.

The applications that are conspicuous by their absence from the above discussion are general dictation such as book writing, and programming. These applications require mass text entry. Mass text entry is much more efficient on a keyboard and seldom requires mobility. Therefore, although the InfoPad system design does not preclude running such applications an add-on keyboard would be required. The issues relating to the use of recognition

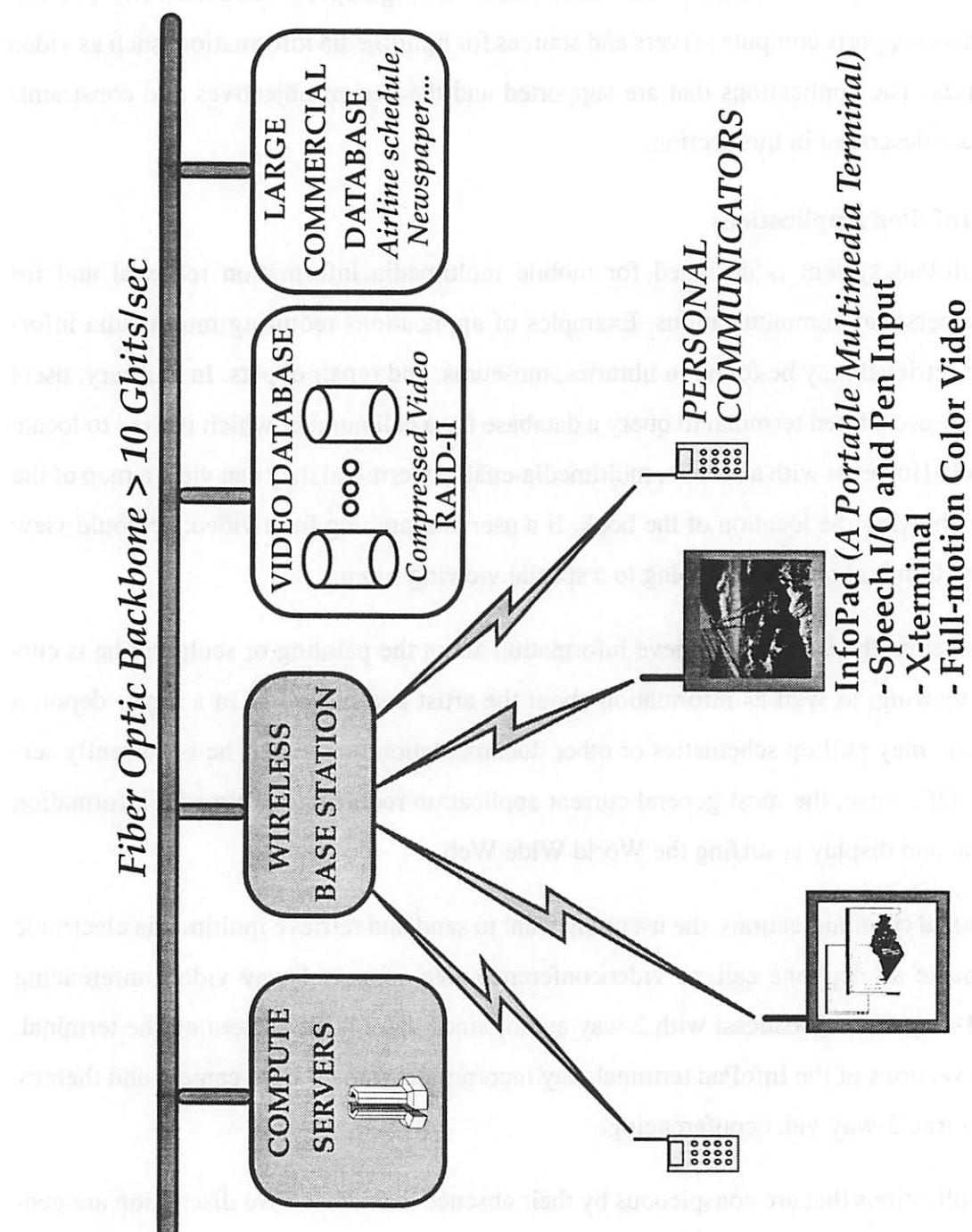


Figure 2-1. Diagrammatic Representation of the InfoPad System

for mass text entry are examined in detail in the discussions on handwriting and speech recognition in Chapter 4 and Chapter 5 respectively.

2.1.2. Design Objectives and Constraints

The primary consideration in the system design is to provide users with a usable terminal. The terminal must be small, light, low-power (to operate as long as possible on the same set of batteries), able to display multimedia (text, graphics, audio and video), and communicate multimedia data in real time. It may also take advantage of the network and compute servers to provide a good user interface and to access data. Ideally, the system should degrade gracefully as its capacity is approached and exceeded. It should also be scalable to allow support of a larger number of users.

Several constraints affect the design. We use commercial technology for the pen digitizer, screen, and battery. We are also constrained by the fact that radio links are inherently error-prone so the system must tolerate missing data packets, which affects the protocols used to communicate data.

2.1.3. Design Choices

Given the above objectives and constraints, the system design concentrated on making the terminal small and light, with the longest battery life possible. The most visible design choice affects the input modalities. Pen and audio input replace the keyboard and mouse. Audio is supported via a head mounted microphone and earphone to reduce audio sensitivity to ambient noise and to limit noise generation. A head mounted microphone helps the speech recognizer but is cumbersome to use. The long term solution is to mount the microphone on the terminal itself and to use better speech recognition algorithms which are more tolerant of noise and channel distortion.

Given the high-bandwidth network connection and the networked compute servers, it is possible to off-load all user-level computation onto the network and thereby minimize the required compute power on the terminal. This is taken a step further by moving all system level computation onto the network as well. Therefore all user and system processes in the

InfoPad system are executed on remote compute servers. The terminal is simply a dumb input-output terminal, reducing cost, weight and power consumption.

To support ubiquitous computing with a scalable solution, the system is designed with a pico-cellular wireless architecture and uses very small cells about 10 meters in diameter. The goal is for each cell to support up to 50 users simultaneously. This operates in an indoor environment where it is possible to place antennas at very small intervals.

Given that the two input modalities are pen (4 kbytes/s) and audio (64 kbytes/s for 8-bit μ -law at 8 kHz), the uplink radio needs to support only 68 kbytes/s per user. This may be implemented using a standard Time Division Multiplexing (TDM) scheme. On the downlink, however, text and graphics must also be supported, requiring a 1Mbit/sec downlink connection. We use a direct sequence spread spectrum scheme on the downlink.

2.2. Wired Network Architecture

Several components, collectively called InfoNet, comprise the wired network software architecture [Le95] (see Figure 2-2).

On the uplink, Gateway A receives data from the antenna for all InfoPad terminals within Cell A. This data is made available to the Pad Server of each terminal on a unique Internet socket. Each Pad Server, which may run on a compute server remote from the Gateway, receives a single bit-stream from the Gateway and de-multiplexes this stream into pen and audio data. This data is made available on unique Internet sockets, as illustrated in Figure 2-3. The socket connections are encapsulated within the InfoNet API.

The Pen Server and Audio Server connect to the pen and audio ports of the Pad Server, and clients may access pen and audio data via these servers. The Pen Server and Audio Server are described in greater detail in Section 3.3 and Section 3.4 respectively.

On the downlink, the Pad Server receives audio data from the Audio Server on the same Internet socket as it sends uplink audio data. The Pad Server also collects text/graphics data and video data from the associated Servers. The Text/Graphics server is a modified X server [Sche91] while the Video Server is a modified Continuous Media Server [Rowe92].

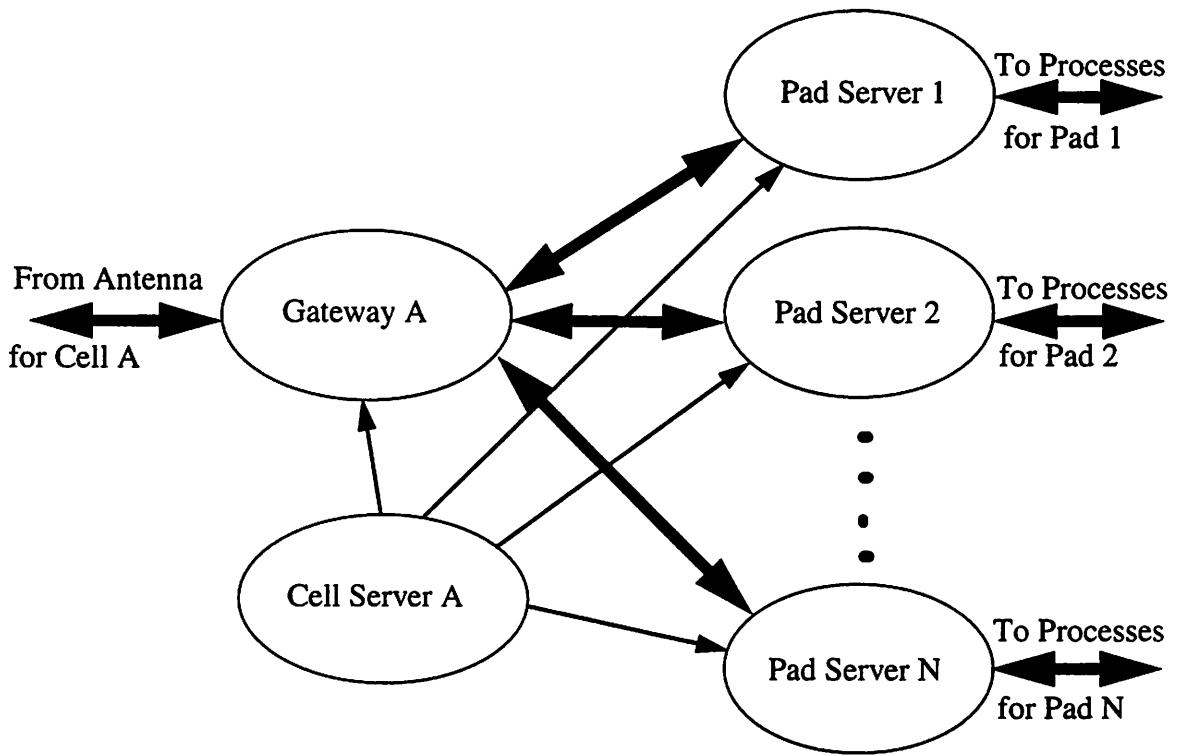


Figure 2-2. The InfoPad Network (InfoNet) Architecture

The Pad Server multiplexes all downlink data into a single bit-stream which it sends to the Gateway.

The Cell Server monitors the cell and facilitates handoff when a user moves between cells [Le95].

2.3. Terminal Design

Two versions of the InfoPad terminal were built. Both use a Sharp LM64P80 LCD monochrome display for text and graphics, and a Logitech Gazelle pen digitizer. Both also use a Sharp 4-inch color active matrix display for video, and 8-bit μ -law audio at 8 kHz for both uplink and downlink.

The architecture of the first version, called IPGraphics, is shown in Figure 2-4 [Chan94]. The Protocol Chip does all the multiplexing and de-multiplexing of data from the various sources and to the various sinks. There is no common data bus in this architecture, so

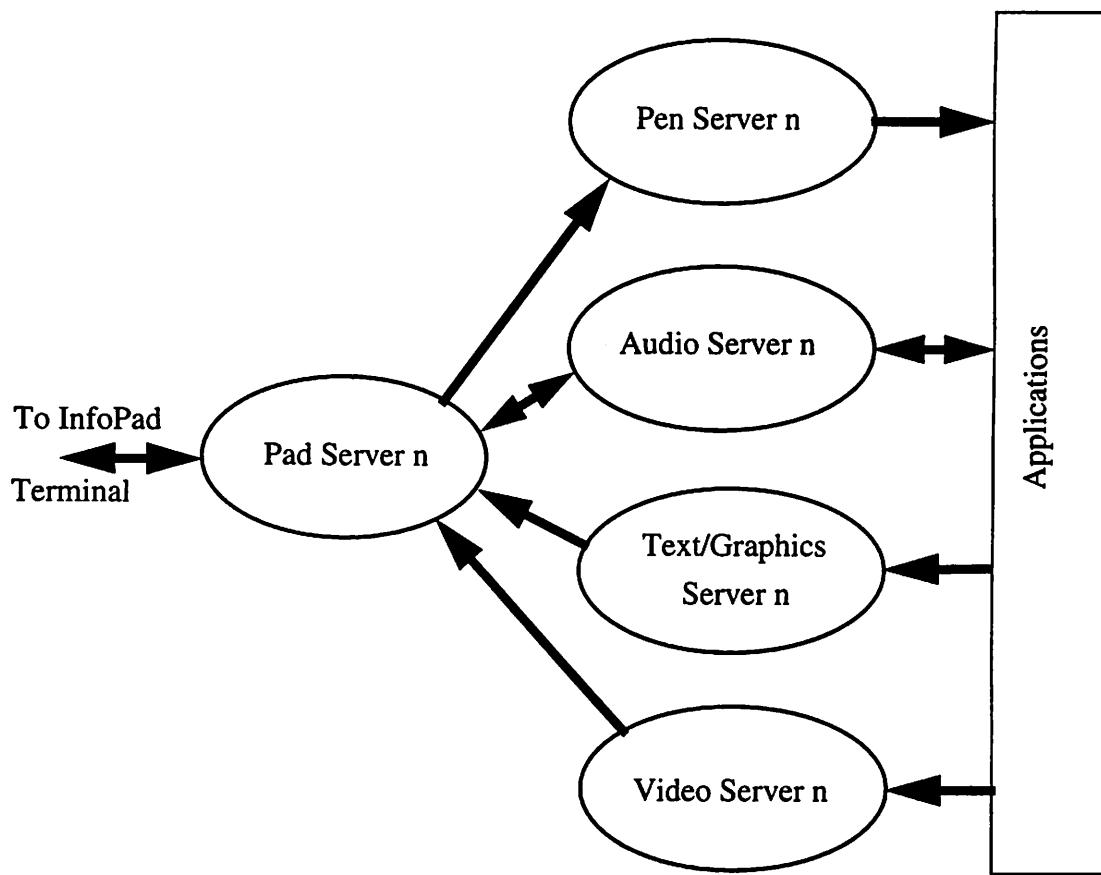


Figure 2-3. InfoPad Type Servers

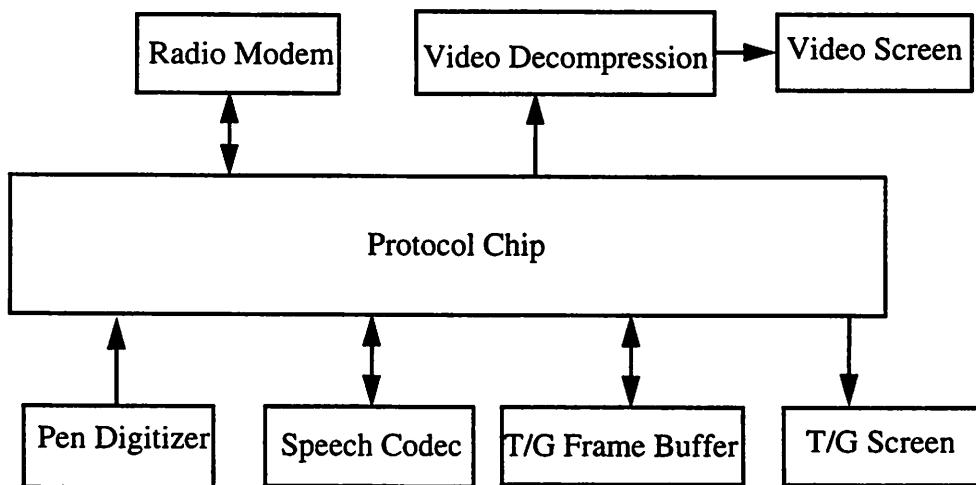


Figure 2-4. Architecture of the IPGraphics Terminal

adding new devices is not possible. However the hardware complexity, power consumption and cost are very low.

In IPGraphics, the Pen Subsystem consists of the digitizer, two commercial chips and some custom circuitry on the Protocol Chip. The Subsystem is described in greater detail in Section 2.4.

The architecture of the second version of the InfoPad terminal, called IPVideo, is shown in Figure 2-5 [Doer96]. A low-power data bus is used for all data communications. All the

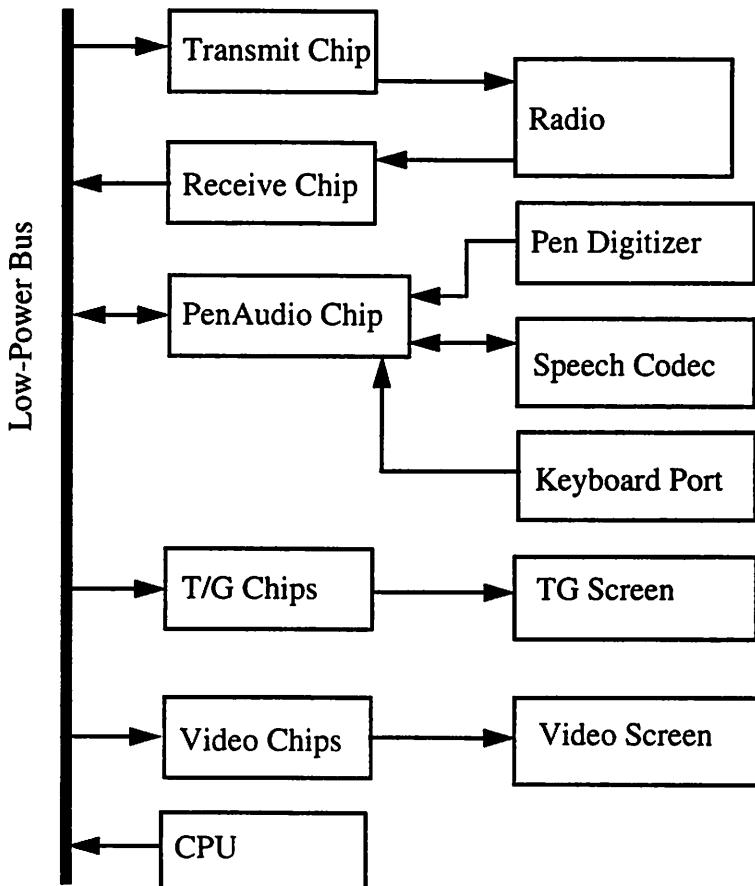


Figure 2-5. Architecture of the IPVideo Terminal

modules talk to the bus in a priority scheme to avoid bus contention. The bus architecture allows adding new I/O devices to the terminal without re-fabrication of existing custom chips.

In IPVideo, the Pen Subsystem consists of the digitizer and some custom circuitry on the PenAudio chip. The external components from IPGraphics are replaced by custom hardware for lower power and smaller board area. The keyboard port and the entire Keyboard

Subsystem are very similar to their pen counterparts. The Pen and Keyboard Subsystems are described in greater detail in Section 2.5.

2.4. Pen Subsystem on IPGraphics

The Pen Subsystem of the IPGraphics terminal is illustrated in Figure 2-6. Its function is

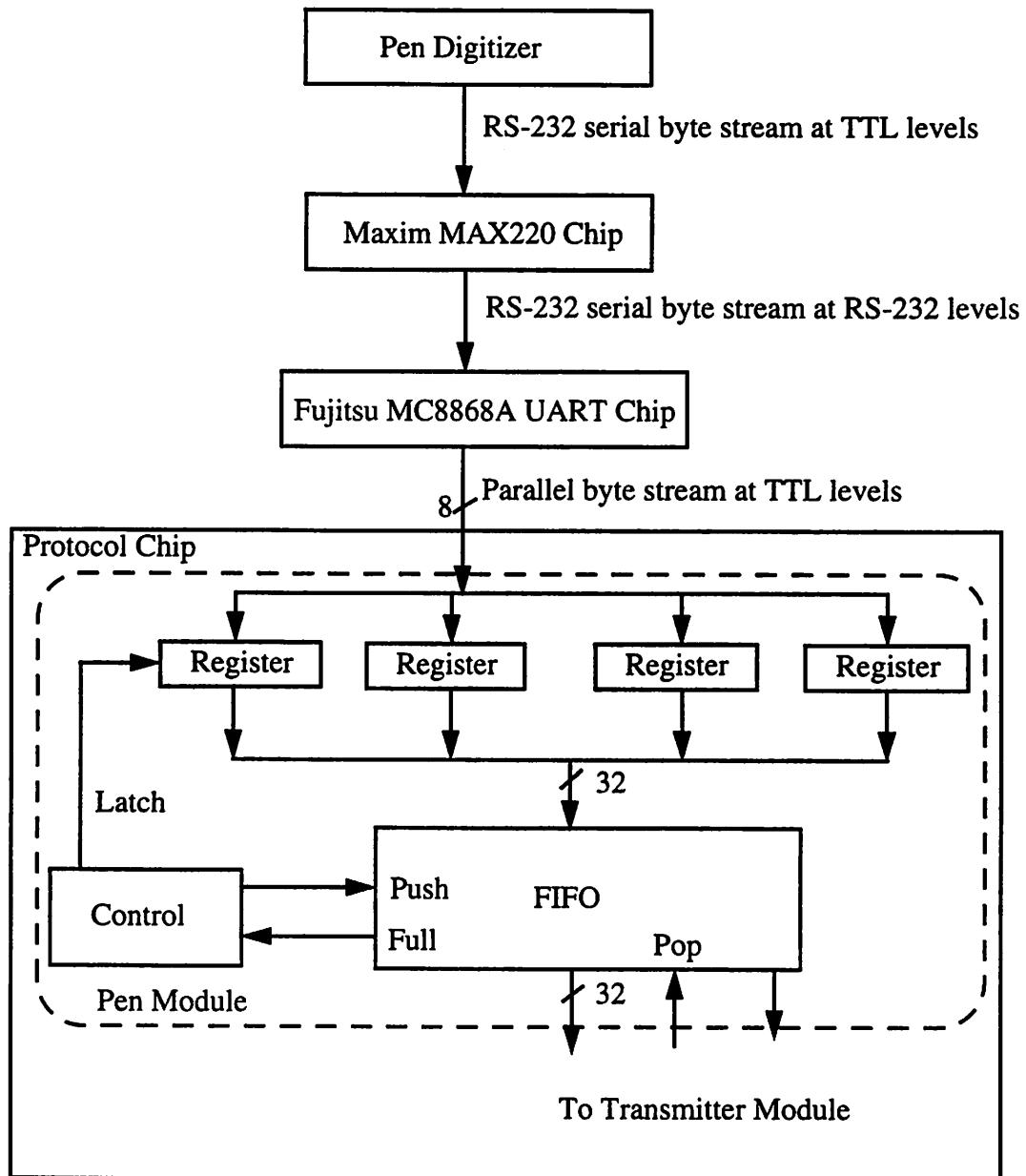


Figure 2-6. IPGraphics Pen Subsystem

to collect pen data from the digitizer, push it onto a FIFO, and signal the transmit chip to send the data from the FIFO to the Pen Server over the radio link.

The pen digitizer provides data as a 9,600 bits/sec RS-232 serial byte stream at TTL signal levels. The MAX220 chip converts TTL levels to RS-232 levels which the MC8868A chip

converts into a TTL 8-bit parallel data stream. This parallel stream is read by the Protocol Chip. On-chip, the Pen Module converts the 8-bit parallel data stream to a 32-bit parallel stream, which is pushed onto a 32-bit, 16-word FIFO. This FIFO is read by the transmitter module on the Protocol Chip.

The interface to the Pen Module is illustrated in Table 2-1. SCAN, SCANIN and

Signal Name	Direction	Source/Destination	Comment
SCAN	in	off-chip	for testing only
SCANIN	in	off-chip	for testing only
SCANOUT	out	off-chip	for testing only
DR	in	UART	Data Ready
RS2323Data	in	UART	Parallel data in
DRR	out	UART	Data Ready Reset
FifoPush	out	FIFO	Push 32-bit data onto FIFO
FifoPop	in	Transmitter module	Pop 32-bit data from FIFO
FifoOut	out	Transmitter module	Parallel Data Out

Table 2-1. Signal Pins on the Pen Module

SCANOUT are for scanpath testing only. **DR** tells the Pen Module that the UART has data that is ready to be read. **DRR** tells the UART that the Pen Module has read the current data. **FifoPush** tells the FIFO to read the current data. **FifoPop** is generated by the Transmitter Module and pops data from the FIFO. **FifoOut** is the 32-bit data that is made available to the Transmitter Module by the FIFO.

A photograph of the Protocol Chip is shown in Figure 2-7. The Pen Module is marked in the upper right of the photograph.

2.5. Pen and Keyboard Subsystems on IPVideo

The Pen and Keyboard Subsystems on IPVideo are shown in Figure 2-8. The function of the Pen Subsystem is to collect pen data from the digitizer and make it available to the Transmit Chip on the InfoPad Low Power Bus. The Keyboard Subsystem performs a similar function for keyboard data.

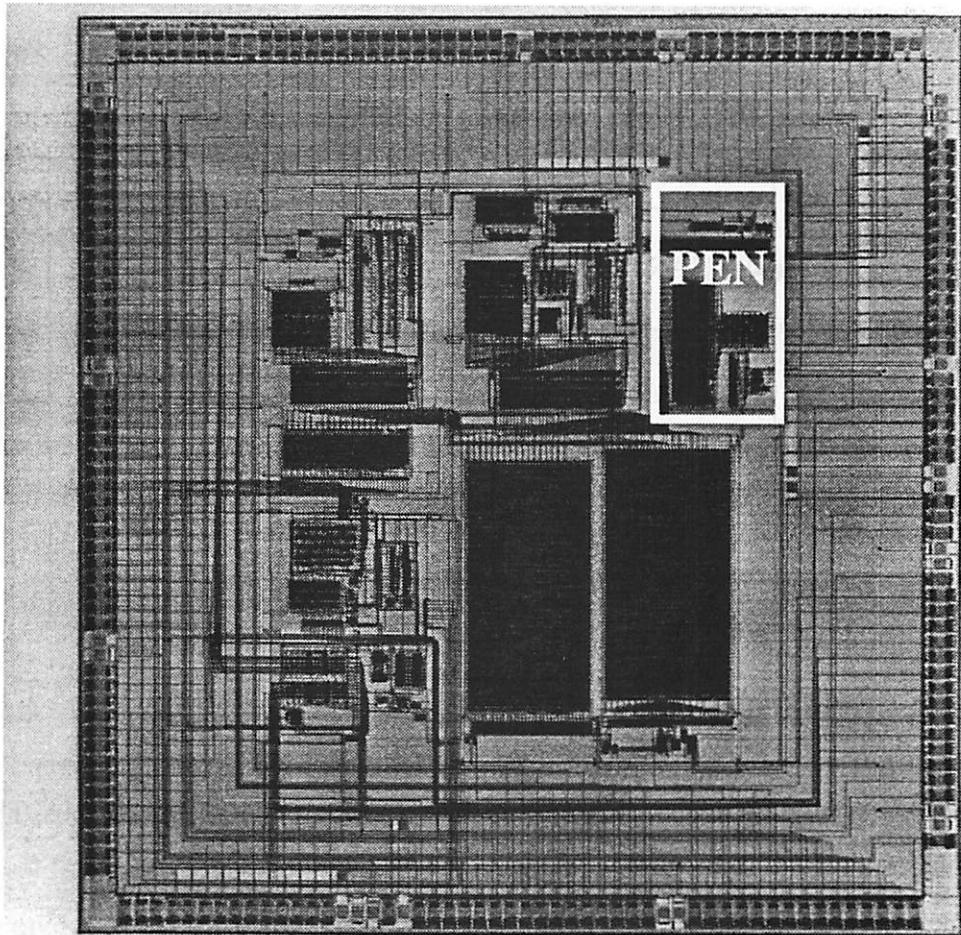


Figure 2-7. Protocol Chip Photograph

In IPVideo, the functionality of the MAX220 and the MC8868A UART are combined into an on-chip UART in the PenAudio chip. Two UARTs take serial data streams at 9,600 bits/sec from the Pen Digitizer and 14,400 kbits/sec from the keyboard respectively and converts them into 8 bit parallel streams. The parallel streams are pushed onto FIFOs which are read by the Transmit Chip via the Low-Power Bus and the BusMaster module.

The BusMaster module is shown in Figure 2-9 and handles the communications protocol with the bus. The bus uses a priority daisy chain scheme. Permission to write to the bus may be requested and granted via the daisy chain. The Request module arbitrates between requests from within the chip and requests from other chips. If the chip wants to control the bus, chips lower in the priority queue must wait until it is serviced.

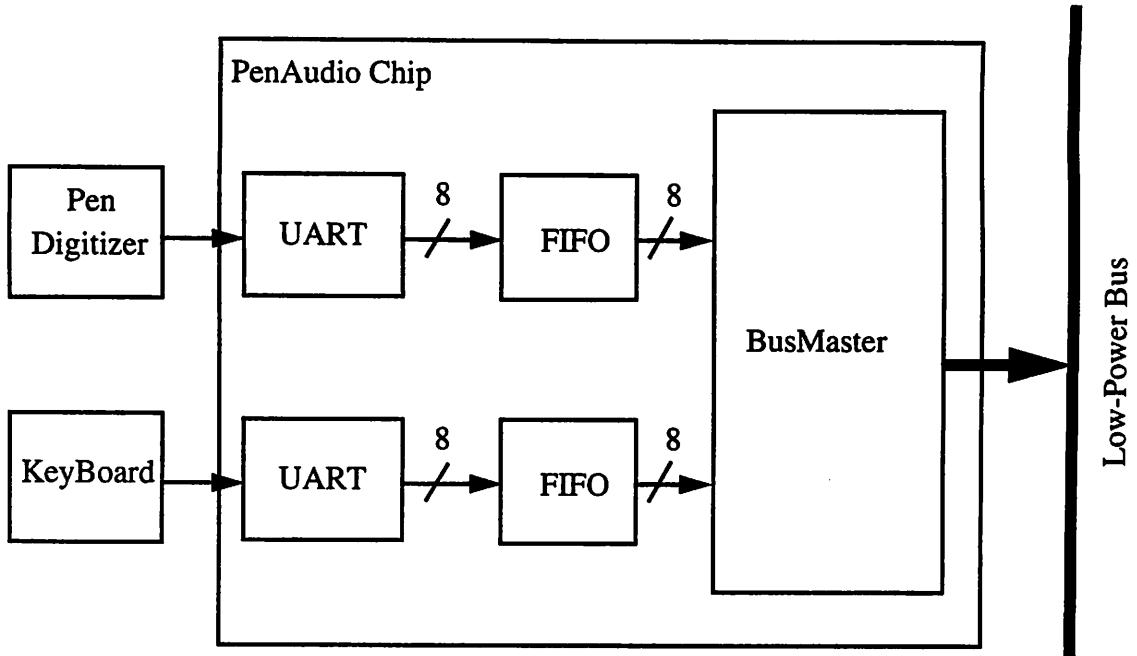


Figure 2-8. IPVideo Pen and Keyboard Subsystems

The Master module takes data from the Packetizer and puts it onto the bus. The packetizer manages the FIFO associated with the appropriate input device and assembles this data into packets. Keyboard data has 1-byte packets while pen data comes in 5-byte packets. The Slave module allows the IPVideo terminal's ARM 610 CPU to program all the registers on the other modules inside the BusMaster Module.

For more details on each component in the BusMaster module, refer to [Doer96].

The on-chip UART is a modified version of a UART created by Brian Richards. A schematic of the UART is shown in Figure 2-10. It takes a clock on the **16XCLOCK** pin at 16 times the bit-rate of the incoming serial stream, just like commercial UART chips. This clock is used to sample the serial stream coming in on the **RXDATA** pin and to detect edges within this stream. Based on these edges, bit boundaries are determined and the incoming bit sequence is detected. The bit sequence is made available in parallel on the **D[7:0]** bus. **VALIDDATA** indicates that there was no parity error and **FRAMEERROR** indicates that there was an error in detecting the stop bits in the serial stream.

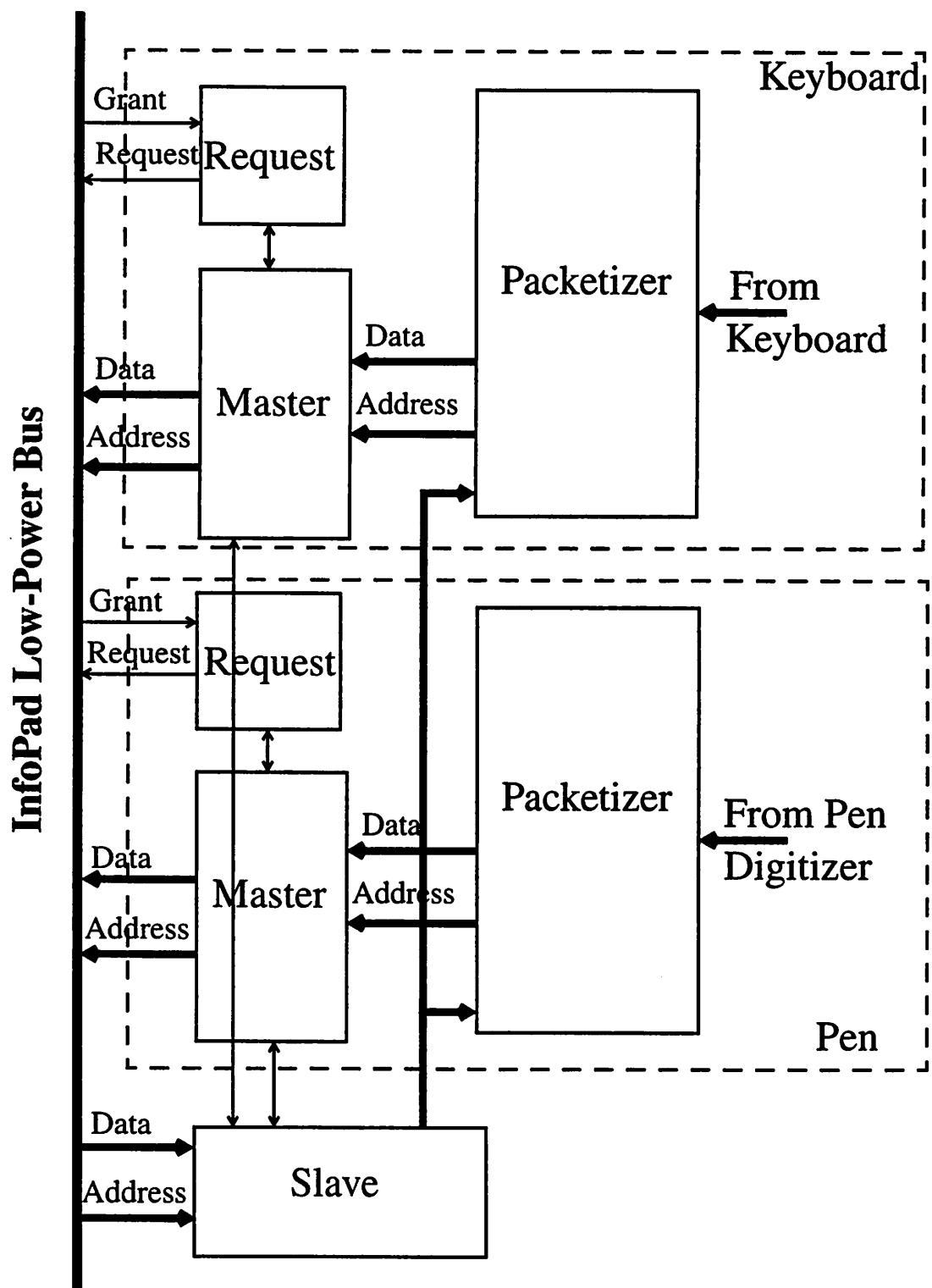


Figure 2-9. BusMaster Module for IPVideo Pen and Keyboard Subsystem

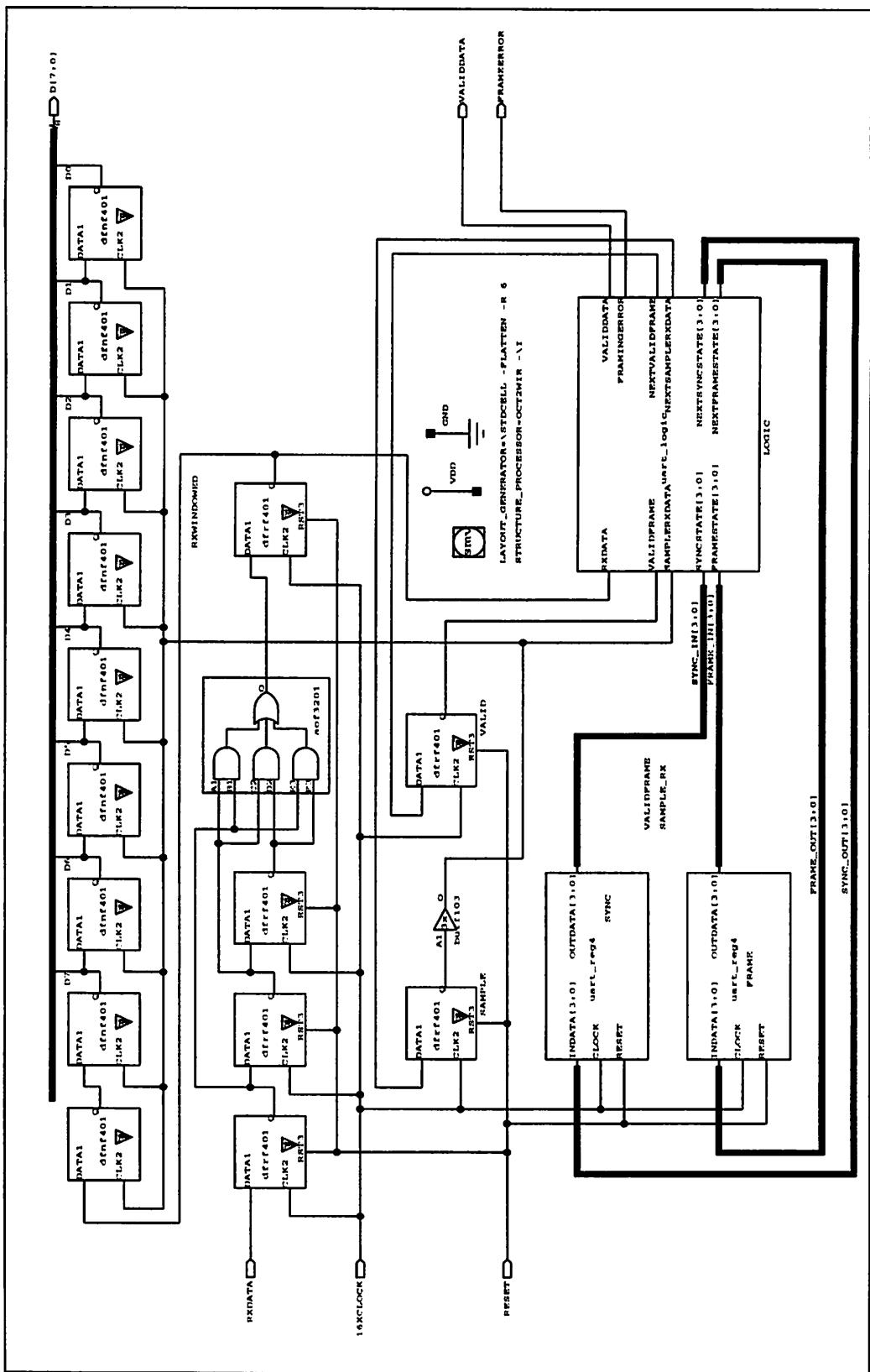


Figure 2-10. Schematic of UART Used in Pen and Keyboard Module

A photograph of the PenAudio Chip is shown in Figure 2-11. The Pen and Keyboard

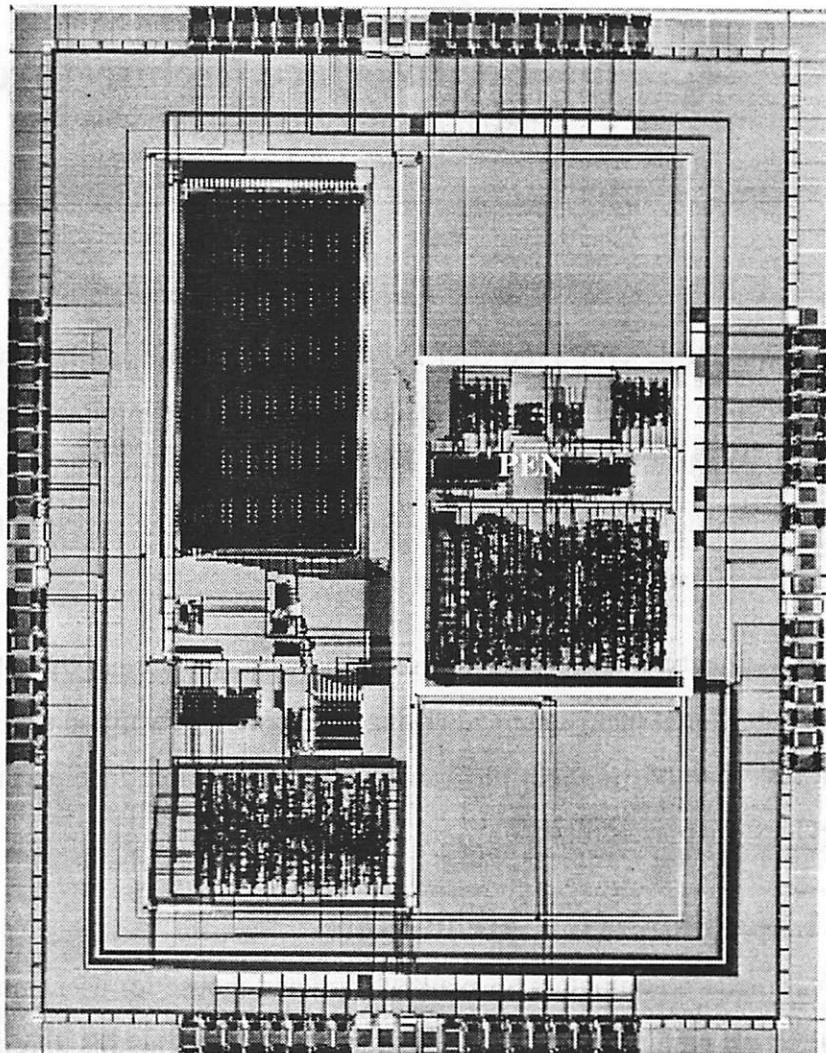


Figure 2-11. PenAudio Chip Photograph

Module is marked in the right half of the photograph.

3 The User Interface Architecture

This chapter describes the User Interface Architecture and its major components. A new architecture is required due to the unique nature of the InfoPad terminal's input modalities. The pen and audio data types are not standard on existing computers or terminals, and the recognized-text data type is new. The new architecture supports these data types and the transfer of data to and from applications.

The architecture takes advantage of the networked nature of the InfoPad system to provide pen data, audio data, and handwriting and speech recognition services to applications. We address mainly the input mechanisms of the user interface and assume standard output devices (display and audio).

3.1. Need for a New Architecture

Most modern computers have a keyboard and mouse for user input. More recently, some portable computers have added pen input, usually using this to replace rather than supplement the keyboard and mouse. There are currently no UNIX and X based systems which use pen input. There are also no commercial products which use audio input, with or without pen, to replace the keyboard and mouse.

On UNIX-X systems, applications see the X server as the sole controlling entity that owns the keyboard and mouse. There are two input event types, keyboard events and mouse events, with other event types (such as pointer entry and pointer leaving events) derived from these types. Correspondingly, there are two data types, ASCII characters and mouse pixel coordinate pairs (with mouse button status). There are well-established conventions for dealing with these event types and data types. Applications receive keyboard events as ASCII character strings. Mouse events may be received as individual XEvent structures or

at a higher level of abstraction, depending on the widget set and other software libraries used. Being network-aware, X supports distributed processing by allowing applications to run on one machine and use the display from another seamlessly. This model of distributing computation is well suited to the InfoPad system.

However, the current X architecture does not allow easy addition of other input modalities. There is no support for pen data, audio data, or recognized text. Nor is there support for incorporating type-transcoders (such as recognizers). Another weakness is that all data types come through a single server, the X server. This is adequate for low rate data sources but for audio, at least 64 kbits per second flows through the server in each direction. This can seriously impact delivery of other data types through the single-threaded server.

We need an architecture that separates the different types of data into independent streams controlled by servers that have separate threads. In our implementation, we do not have access to a multi-threaded operating system so each data type server runs as a separate process. We also need each server to be transparently network-aware so that we can distribute the computational overhead of the User Interface Architecture. This is especially true given the computational demands of handwriting and speech recognizers.

3.2. High Level Description

The User Interface Architecture is shown in Figure 3-1. Its major components are the Pen Server, Audio Server, handwriting recognizer, and speech recognizer. The Pen Server reads a raw pen byte stream from the Pad Server, translates it into a more tractable form as described in Section 3.3 below, and makes this data available to applications that require pen-resolution data. It also emulates the mouse by generating X-windows mouse events so that pointing functions are supported without applications having to access the Pen Server directly.

The Audio Server reads a raw audio stream from the Pad Server, buffers it and makes it available to applications. It also reads audio data from applications and sends this data to the Pad Server. The Audio Server is described in greater detail in Section 3.4. The handwriting recognizer reads raw pen data from the Pen Server or from applications and gener-

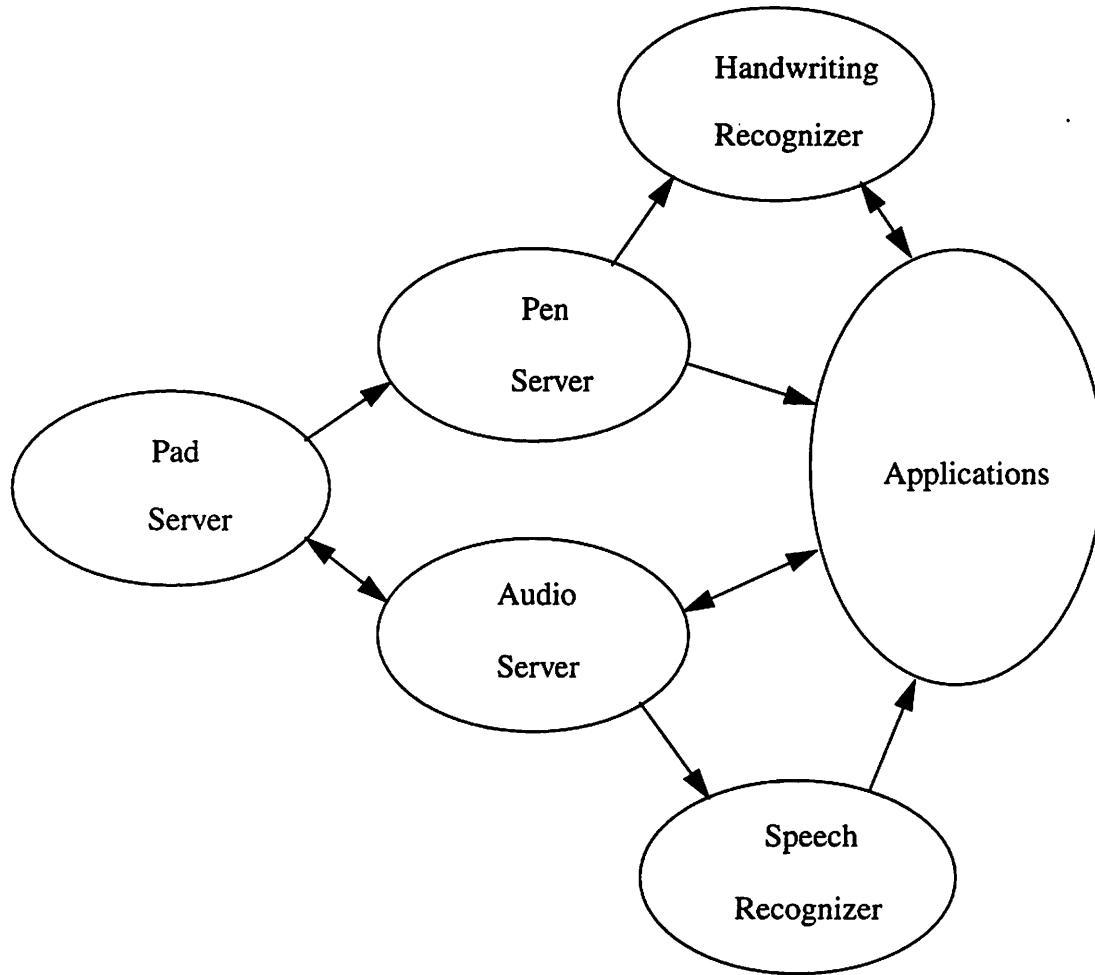


Figure 3-1. InfoPad User Interface Architecture

ates recognized text or symbols as appropriate. The speech recognizer reads audio data from the Audio Server and generates recognized text. Each recognizer pictured in the Figure may be a collection of one or more specialized recognizers, as explained in Section 3.5.1 below.

In Figure 3-1, raw data is shown going from the data type servers to the recognizers. However, there are situations where data is collected by the application itself and sent to a recognizer. This situation is described in greater detail in the chapters on recognition, Chapter 4 and Chapter 5.

Each component communicates with the others through Applications Programming Interfaces (APIs) which encapsulate the Internet socket used for inter-process communication and the locations of the other components. This allows the programmer to work at a level of abstraction which does not require knowledge of socket communications nor knowledge of the location of each component.

The use of Internet sockets for communications allows each component to run on a different machine, thereby distributing the computation onto multiple processors. This allows each component to run without interference due to CPU usage by other components. This is particularly important since some components, such as handwriting and speech recognizers, are compute-intensive and other components, such as the Audio Server, are very sensitive to latency. Distributing the processes also allows services to be provided by alternative vendors if the communications protocols remain the same. It is therefore possible to upgrade recognizers transparently without recompiling applications or other infrastructural code.

Each component of the user interface has its own API. As far as possible, we make use of existing standard APIs but in some cases we designed our own interfaces in the absence of existing standards.

There are a few disadvantages to this architecture. Firstly, the communications overhead of the Internet sockets adds about 2ms per hop [Nara96]. We could reduce latency by combining all the type servers, including recognizers, into the Pad Server. This solution will become practical if we use a multi-threaded operating system and powerful compute servers that can handle all the threads concurrently.

However the 2ms overhead is a small price to pay for the advantages we gain from multi-server operation. There is a minimum latency below which the system does not benefit from a further latency reduction. This lower latency limit is 30 msec, the screen refresh period, and arises due to the fact that system feedback cannot be displayed sooner than 30 msec after the last screen update. There is also a maximum latency that can be tolerated, which is determined by perceptual factors. Delays of less than 100 msec are not usually noticed by users, and delays of greater than 200 msec are usually noticed.

Secondly, each read from and write to an Internet socket is a system call, which consumes a significant amount of CPU time. Our network system's main bottleneck is currently the number of system calls it takes to run the system. However, this bottleneck will also go away when we move to a multi-threaded operating system and use shared memory.

Thirdly, the distributed nature of the architecture requires that a database be maintained to keep track of the location of each service. This database manager, called the Name Server, maintains a list of currently available services and their locations on the Internet. The Name Server is centralized and therefore does not scale well as the system grows. The solution to the scalability problem is to distribute the database, keeping the entries on Name Servers that are local to each network cluster.

Lastly, each network hop consumes network bandwidth, which reduces the number of terminals the system can support. However, with proper network design this should not be a limiting factor.

Now that we have described the User Interface Architecture at a high level, we will describe its individual components in detail.

3.3. Pen Server

The Pen Server controls access to a scarce resource, the pen digitizer port. It performs two functions. Firstly, it emulates the mouse by sending pointer events to the X server in screen-resolution coordinates, allowing the user to run standard X applications that use the mouse for pointer input. Secondly, it makes pen-resolution pen data available to applications that require data with that resolution, such as handwriting recognizers. This section describes the design and implementation of the Pen Server.

3.3.1. Mouse Emulation

The Xlib software library allows applications to send input events to the X server. These events include keyboard events, pointer motion events and pointer button events. However, many X window managers handle pointer button events specially by trapping them and grabbing further pointer button events directly from the mouse port. This effect prevents the Pen Server from fully emulating the mouse since the X server effectively freezes

every time a mouse button is pressed. In order to completely implement the emulation, the X extensions were used. The extensions allow the Pen Server to circumvent the limitations of the Xlib interface by accessing the X server at a lower level.

There are three extensions that would work in this case, the XInput extension, the XTEST extension, and the DEC-XTRAP extension. We use the DEC-XTRAP extension.

3.3.2. Pen-Resolution Data

The InfoPad Pen Server receives a raw pen byte stream from the pen digitizer via the radio and InfoNet, without translation. This byte stream is interpreted by the Pen Server and pen resolution data is made available to applications.

The current implementation of the InfoPad terminal uses a Logitech Gazelle digitizer which provides a 5-byte packet (see Table 3-1) encoding pen position, button status and

Byte No.	MSB 7	6	5	4	3	2	1	LSB 0
1	1	Near	Battery	Stat3	Stat2	Stat1	Barrel	Tip
2	0	X6	X5	X4	X3	X2	X1	X0
3	0	0	0	X11	X10	X9	X8	X7
4	0	Y6	Y5	Y4	Y3	Y2	Y1	Y0
5	0	0	0	Y11	Y10	Y9	Y8	Y7

Table 3-1. Pen Packet Structure for the Logitech Gazelle Digitizer

other information. **Near** indicates that the pen is near enough to the digitizer that the position and button status bits are valid. **Battery** indicates pen battery status. **Stat1-3** reflect data bits set in hardware; they are currently set to 0. **Barrel** and **Tip** report the status of the barrel and tip buttons respectively. **X0-11** and **Y0-11** are pen x and y coordinates respectively. The origin for **X** and **Y** is at the top-right corner of the digitizer (and therefore of the screen).

However, future versions of the terminal may use digitizer hardware from other vendors so the Pen Server supports several commercial digitizers, including two types of Wacom digitizer and the Scriptel digitizer. The choice of digitizer decoding is specified on the

command line. Support for other digitizers may be easily added with minor modifications to the software.

Most pen digitizers provide data at a resolution of 200 lines/inch which results in about 2700 x 1900 pixels for the Gazelle digitizer, whereas the InfoPad terminal has a 640 x 480 pixel screen, about a factor of four difference. To support future versions of the digitizer which might have even greater spatial resolution, and to provide digitizer independence, the Pen Server provides pen data at 8 times the screen resolution. The translated pen packet that is provided to applications therefore contains x and y coordinates in both pen and screen resolution, pen button status, and a timestamp (see Figure 3-2). The coordinate ref-

```
typedef struct tablet_packet {
    char head;
    char buttons;
    short x;
    short y;
    short xscreen;
    short yscreen;
    unsigned long timestamp;
} tablet_packet;
```

Figure 3-2. Pen Server Packet Structure

erence frame has its origin at the top-left corner of the screen, with (0,0) at pen resolution exactly corresponding to (0,0) at screen resolution, independent of the digitizer used.

The Pen Server is the only place in the entire system, including both hardware and software components, where knowledge of the pen digitizer vendor is required. Its interfaces at both ends use protocols which are independent of digitizer vendor. Therefore upgrades to the digitizer hardware require changes in the Pen Server and nowhere else.

3.3.3. General Operation

The Pen Server is started by the Pad Server when the user first turns the InfoPad terminal on. It runs on a machine whose address is determined via the InfoNet Name Server.

On start-up, the Pen Server establishes a connection to the X server and the DEC-XTRAP extension, and establishes a connection to the Pad Server, which is part of the InfoNet soft-

ware (see section [Section 2.2.](#) on page 16). It creates a pair of Internet sockets, one for data and one for control, which are advertised via the Name Server. Applications may obtain pen-resolution data via these sockets. There is no predefined limit to the number of applications that may concurrently connect to the Pen Server and receive pen-resolution data. Although allowing multiple clients to access pen data without security may seem dangerous, it allows us to monitor the performance of the Pen Server and also allows the handwriting recognizer to snoop on the pen data if it chooses to read this data directly from the Pen Server rather than relying on the application to provide it.

The pen byte stream is read from the Pad Server and the necessary translations and X operations are performed. Pen resolution data is calculated but not stored since there are no client connections from applications. Once an application connects to the Pen Server, translated pen packets are sent to the application. Each application uses the Pen Server's API (see [Section 3.3.5](#)) which allows the application to establish and close connections, pause and resume data transfer, and control the type of data sent. For example, an application may prefer to receive only pen-down packets, whereas another application may prefer pen-down packets as well as the first pen-up packet after pen-down, allowing detection of pen lift events. Since there is no standard for such an interface, we designed our own API that suited our needs.

The Gazelle digitizer provides pen data at about 100 pixels/sec, much faster than the rate of a typical mouse. This is acceptable for handwriting recognition and other applications requiring pen-resolution data, but the high data rate can overwhelm the X server. Therefore, the Pen Server temporally sub-samples the events it sends to the X server. This has the effect of making X-aware applications receive exactly as many pointer events as though they are being driven by a mouse. However this reduced resolution can hurt handwriting recognition accuracy. Applications such as handwriting recognizers which cannot tolerate sub-sampled data connect directly to the Pen Server over Internet sockets via its API to obtain full-resolution data.

3.3.4. Command Line Options

Several command-line options allow the InfoNet system or other calling program to customize the Pen Server on system start-up. These options allow the user to specify:

- display screen to use for X events (defaults to the UNIX XDISPLAY environment variable)
- ID of the current pad: the Pen Server connects to this terminal via InfoNet
- Internet port number on which to make pen-resolution data available
- device name of the serial port (such as /dev/ttya) from which to read raw pen data instead of InfoNet
- tablet type: allows selection from Logitech Gazelle, Wacom SD-510, Wacom UD-0607 or Scriptel tablet

For more details, refer to the manual page in the InfoPad system documentation.

3.3.5. Applications Programming Interface

Applications that want access to pen data can use the C language API. This section lists and explains the functions that comprise this simple interface. In the text below, **penStatus** is an enumerated type that can take on values **PS_OK** and **PS_ERROR**.

PS_OpenConnection() opens a connection to the Pen Server that services this InfoPad terminal and returns a **penConnection** structure that is used for all further references to the Pen Server:

```
penConnection *PS_OpenConnection (char *host, int port,  
                                int padID)
```

If an error occurs, an error message is printed and NULL is returned. If **padID** is given, that number is used to query the Name Server to determine the location of the Pen Server. If **padID** is -1, then **host** and **port** are used instead.

PS_CloseConnection() closes the connection to the Pen Server and frees all memory and state associated with the **penConnection** object:

```
penStatus PS_CloseConnection (penConnection *conn)
```

PS_GetFD() returns an integer file descriptor for the pen data socket:

```
int PS_GetFd (penConnection *conn)
```

The file descriptor is used by the application to determine whether pen data is waiting to be read. This is done via the select() system call.

After calling PS_OpenConnection(), the application must call PS_SendStart() to signal the Pen Server to start sending pen data:

```
penStatus PS_SendStart (penConnection *conn)
```

PS_SendStop() tells the Pen Server to suspend sending data to the application. However it does not flush the data connection:

```
penStatus PS_SendStop (penConnection *conn)
```

PS_GetPacket() reads a single pen packet that was sent by the Pen Server:

```
penStatus PS_GetPacket (penConnection *conn,
    int *x, int *y, int *b1, int *b2,
    int *xscreen, int *yscreen, int timestamp)
```

PS_Flush() tells the Pen Server to suspend sending data to the application and also flushes the pen data connection so that a subsequent call to PS_SendStart() followed by calls to PS_GetPacket will not return obsolete data:

```
penStatus PS_Flush (penConnection *conn)
```

There are also other functions which tell the Pen Server whether to send pen data for pen-up strokes.

3.3.6. Pen Support for X Workstations

The Pen Server can also run independently of the InfoPad system. This allows software development to proceed while the InfoNet system and the InfoPad terminal itself are not available. In this case, the Pen Server does not connect to the InfoNet Pad Server but rather to the serial port of the workstation in order to read pen data. This requires that a pen digitizer of one of the supported types be connected to that port. It is assumed that the digitizer sends data at 9600 baud.

This mode of operation is selected via a command line option.

3.4. Audio Server

The InfoPad Audio Server controls access to the terminal's audio input and output ports. On the uplink, it collects 8-bit μ -law audio data from the InfoPad terminal and maintains a buffer from which applications can read audio data. On the downlink, it accepts 8-bit μ -law data from applications, mixes the data from all incoming streams linearly and sends it to the terminal. The downlink policy is to mix the incoming audio streams rather than a preemptive scheme. This is due to the need for all applications to be able to concurrently play sounds on the speaker. For example, if a video player application is currently playing audio continuously, another application that wants to play a warning beep must be able to play the beep in a timely fashion without interrupting the video player's audio stream.

Since dropouts and other impairments to an audio stream are very audible, care must be taken to ensure the integrity of each audio stream. Therefore, it is essential that audio data be buffered in the Audio Server. The Audio Server maintains a randomly-accessible 16-second buffer for each of the uplink and downlink, which seems sufficient for all of the applications we have built so far.

3.4.1. AudioFile Compatibility

The InfoPad Audio Server is based on AudioFile [Leve93], a public-domain software package that provides network-transparent access to the audio ports of standard UNIX workstations. Its operation is very similar to that of the X server. In X, the X server controls access to the screen, and applications may use the screen resource only via the Xlib interface. Similarly, AudioFile takes control of the audio input and output ports of the workstation and applications may access these resources only through AudioFile.

There is a well-defined, standard API for AudioFile. The InfoPad Audio Server uses the same API library. This allows all applications that are AudioFile compatible to run on the InfoPad terminal without modification or even recompiling. Some important applications that support AudioFile include the MBone videoconferencing tools [MBo95].

Using a standard, widely supported interface also allows software development to proceed before the InfoNet software and InfoPad terminal are available. Application developers

need only run the standard AudioFile server on their workstations to be able to test their software.

3.4.2. Enhancements Over AudioFile

Internally, the Audio Server differs from AudioFile in several ways. The device-dependent code that reads audio data connects to the InfoNet Pad Server rather than the workstation's audio hardware port. Writing of audio data also goes through the Pad Server.

In AudioFile, timing consistency is maintained by counting the number of bytes read from the audio hardware. There is therefore a common timebase for uplink and downlink. But in the InfoPad, the unreliable radio link means that uplink data may frequently be lost, making this method of timing measurement unreliable. In the Audio Server a separate timing mechanism based on UNIX time is used for the downlink. The uplink and downlink timebases are therefore independent.

The InfoPad terminal's audio chip maintains a small first-in-first-out (FIFO) buffer for the uplink and another for the downlink. The uplink buffer stores 8 bytes (1 ms) of audio data while the downlink buffer stores 128 bytes (16 ms) of audio data. It is necessary to keep these buffers small to satisfy telephone or other applications that require a low-latency audio loop. The small size of the downlink buffer means that data must be sent to the audio chip at a carefully measured rate to avoid overflowing or underflowing the buffer. This rate control is done by the Audio Server. It sends audio data on the downlink in 80-byte packets at intervals of 10 ms. Occasionally, the buffer may indeed underflow if the Audio Server swaps out; due to the rate control on the downlink the effect is only a single click. This underflow problem can be avoided in future by using a real-time operating system and by pinning the Audio Server to main memory.

The Audio Server connects to the InfoNet Pad Server on one side and to applications on the other side via Internet sockets; it is truly networked. It can run on any Internet-enabled machine whereas the traditional AudioFile implementation requires that the server run on the physical machine upon which the audio hardware resides.

3.5. Handwriting and Speech Recognizers

In the sections above, the tasks of the Pen Server and Audio Server are shown to be very well defined. However the tasks of the handwriting and speech recognizers are not as well defined. Loosely speaking, they provide applications with ASCII text which may be used for commands and control or for data entry. However this text is generated from the recognizers' interpretations of handwritten or spoken input and these interpretations may not be accurate. This section addresses the role of handwriting and speech recognition in the InfoPad system and leaves detailed discussion of the actual recognizer designs and implementations to Chapter 4 and Chapter 5 respectively.

3.5.1. Handwriting Recognizer

The handwriting recognizer's function is to replace the keyboard for text entry and for single-key commands. This is a service which all applications requiring textual entry would use. However, there are several different kinds of recognition services which applications may want from the handwriting recognizer.

For mass text entry, a recognizer with a large vocabulary and a grammar would be ideal. Users would write sentences in a natural or artificial language, such as English or C++. The recognizer would automatically check grammar and spelling. For mass text entry in English, cursive handwriting may be used since cursive recognition requires a dictionary.

For file names or web addresses, a recognizer which does not impose a grammar or dictionary is needed. The user prints each word and the recognizer sends the recognized text to the application. In this case, the recognizer would be independent of the application since there is no application-dependent grammar or vocabulary.

Single-character recognition and gesture recognition have similar requirements. Single characters and gestures allow the user to enter macro commands quickly and easily. In this case, the screen location of the drawn character or gesture must also be sent to the application, together with a ASCII string that represents the character or symbol so that the application can use location information to process the recognized gesture. For example, if the gesture indicates erasure of a window on the screen, location information can identify the appropriate window for erasure.

Using the InfoPad architecture, the application may need to access several recognizers concurrently, one for print and one for gestures in the case of an e-mail application. The API allows the applications programmer to specify the vocabulary and grammar he requires so that the recognizer service can provide the appropriate kind of recognition service. The application may therefore use several recognizers concurrently, each performing a different service. As described in Chapter 4, they can all have the same API.

In the next section, we will see that the speech recognizer can perform some of the functions described above. However, it is necessary that all functionality be supported without speech recognition because speech recognizers usually perform inadequately in noisy environments. There are also many situations where the user must be quiet and therefore cannot speak to the terminal.

3.5.2. Speech Recognizer

The system is usable without speech recognition, since text and commands may be entered using the handwriting recognizer. However the speech recognizer adds a new dimension to the user interface since it is not spatially specific. The speech recognizer does not require any kind of navigation, which means that issuing a command using speech need not interfere with the current task. As in handwriting recognition, there are several kinds of services the speech recognizer can provide.

The speech recognizer may be used for dictation or for commands. In dictation, which includes mass text entry such as book writing and programming, the situation is similar to that of handwriting recognition in that a grammar and vocabulary are required. This grammar and vocabulary may be application-dependent. But in the more common case of general dictation, the vocabulary and grammar are application-independent.

For commands, the grammar is much more restrictive and therefore simpler, although the vocabulary may be large for some applications. As in handwriting recognition, the vocabulary is application-dependent.

Unlike handwriting, speech recognition is not appropriate for specifying data which does not have a predefined vocabulary. Spelling out a word verbally takes much more time than writing it, and is therefore less efficient.

Screen navigation using speech recognition may also be inefficient. If the user intends to merely jump from window to window, speech recognition may be used to identify the name of the destination window provided the windows have unique names, or that the user is satisfied with relative rather than absolute moves. However using the pen would be more efficient so we do not support speech recognition based screen navigation in the InfoPad system.

3.5.3. Servers Versus Software Libraries

From the foregoing, it is clear that each application may require more than one recognition service. However, if each application were to have its own dedicated recognizer for each type of recognition service it requires, there will be a large number of recognizers in the system, consuming system resources. If the recognizers were indeed dedicated, there would be no functional difference if each recognizer were compiled into that application. The main disadvantage of compiling the recognizer into the application is that the computational load of the recognizer would then run locally wherever the application is running. This is not a problem if the application is run on a fast compute server or if the recognizer is sufficiently lightweight, but can be a consideration in the more general case.

Compiling the recognizers into each application would make applications very large. This is wasteful especially since at any one time only one application would be using recognition services anyway. Therefore, recognition service for a single user should be provided by a single remote recognition server rather than by compiling recognizers into applications. Moreover, these servers can be upgraded by service providers as technology improves and better recognizers become available, and may be shared between users in addition to sharing among applications. This is especially true for dictation recognizers, which are large and have high computational demands yet do not require customization for each application.

4 Handwriting Recognition

In this chapter we explore the handwriting recognition needs of the InfoPad system and describe the kinds of handwriting recognizers required to meet these needs. We then describe models for delivering recognized text and an on-line handwritten print recognizer that was designed and implemented as part of this research effort and deployed in the system.

In the following discussion we concentrate on handprint recognition and not cursive recognition. We further specialize to the recognition of isolated printed words rather than sentences since this kind of handwriting recognition is required in the InfoPad system, as explained in Section 4.3 below. All the systems described in this chapter perform on-line handwriting recognition; that is, they receive a sequence of pixels from a pen digitizer while the user is writing. This is distinct from off-line recognition, in which scanned images or facsimiles of handwritten words are processed.

4.1. Previous Work

In this section we review some of the previous work reported in the handwriting recognition literature. We do not have recognition accuracy figures for some of these recognizers since their product literature does not report these statistics.

4.1.1. GO Corporation

GO Corporation built a handprint recognizer as part of their PenPoint operating system. The handprint engine recognizes mixed upper and lowercase letters, numerals, and punctuation. It handles both boxed and lined handwriting and tolerates characters that overlap, touch or share strokes, independently of stroke order and direction.

This recognizer achieves 90-97% character level accuracy [Carr91]. Character recognition is performed by comparing character shapes against a set of character prototypes for each character. Users may train the prototypes as well as add new prototypes via a brief training session. A 100,000-word dictionary, standard punctuation rules and application-supplied word lists help improve word and sentence recognition accuracy.

4.1.2. IBM

Researchers at the IBM T. J. Watson Research Center built an unconstrained handwriting recognizer using a technique called Parallel Dynamic Programming [Fuji93]. A preprocessor breaks up incoming words into sub-character elements which are then used for recognizing words from a prescribed list. Letter models are constructed from these sub-character units and an alignment model bridges characters and words.

This recognizer achieves a writer-independent word recognition accuracy of 89.7% based on a vocabulary of 2171 words. This technique works well when a dictionary is used but may not work well for applications without a dictionary since character segmentation is not done. Without segmentation, the search space explodes since the recognizer receives no hints to help it determine character boundaries.

4.1.3. AT&T

Using Time Delay Neural Networks, researchers at AT&T Bell Laboratories built a writer independent and writer adaptive character recognizer [Guyo92]. Classical back-propagation training for writer independence was combined with postprocessing which allows adaptation to unique writing styles and learning on new symbols.

Input characters are first re-sampled, centered and re-scaled to remove time and space distortions. A set of 7 geometrical features are calculated for each pixel. These features are designed to capture local topological features along the curve of the character. The features are fed into a neural network that is designed to extract more complex, global features.

The recognition accuracy achieved was 97.2% without writer adaptation. With adaptation, this figure improved to more than 98%.

4.1.4. CIC

The Communications Intelligence Corporation sells a handprinted character recognizer which does not require user training yet adapts to the user. It is compatible with Windows for Pen and PenDos, and supports several languages including European languages and Japanese. The product, called Handwriter, allows gesture-based editing and includes a signature verification feature to allow secure access to data. Several PDAs, including the Symbol PPT 4600 and some Fujitsu models, bundle this product. Handwriter can also be purchased as a stand-alone product running under Windows, OS/2 or PenDOS. It includes a 6" x 9" graphics tablet, Pen Extensions for Windows or OS/2, and some pen-enabled applications.

4.1.5. Paragraph

Paragraph International's CalliGrapher [Par96] recognizes printed, unconstrained cursive, and mixed handwriting, and is shipped with the Newton MessagePad. It benefits from user training and can recognize arbitrary sequences of symbols, including word fragments. Currently, CalliGrapher supports, English, French and German handwriting.

4.1.6. Apple

A recent handprint recognizer from Apple uses neural networks to estimate the probability that the handwritten input matches each character [Lyon96]. A number of innovations make this recognizer interesting. It uses negative training to reduce the probability of incorrect segmentation during the early stages of recognition, improving word recognition accuracy. The handwriting data in the training set is warped in several ways and the warped versions are used to train the neural net, improving the generality of the net. Under-represented data in the training set is passed through the net more than once during training (in the warped instantiations) so that infrequently occurring examples are less undertrained. The output error used in running the back-propagation algorithm to train the net is normalized by reducing the back-propagated error for classifier outputs corresponding to the incorrect classes relative to the correct class. This helps deal with the situation where the recognizer erroneously eliminates an alternative because one character has almost zero probability due to being a poor alternative choice.

4.2. Characteristics of Handwriting Recognizers

There are several characteristics of handwriting recognizers that may be traded off against each other to produce the best performance for a given application. In this section we examine these characteristics and the issues surrounding handprint recognizers, including methods of using these characteristics to improve recognition accuracy.

4.2.1. Vocabulary

The imposition of a vocabulary on a handwriting recognizer helps reduce the search space and thereby reduces the recognition error rate. For example, if an unconstrained word-input task has an average word length of 5 characters and the error rate is 10% for each character, the probability of getting the word correct is 59% and the search space is 11.9 million lowercase words. But limiting recognition to a vocabulary of 60,000 words reduces the search space by a factor of 200. The reduction in the size of the search space greatly reduces the processing requirements, in this case by a factor of 200. Recognition accuracy improves as well since there are fewer legal candidates.

Applications can further constrain the search space by specifying the kind of input that the recognizer can expect. Using a recognizer that recognizes cursive, print and gestures at the same time will not be as accurate nor as fast as using separate recognizers for each input type. The best-case scenario is where the application knows which kind of input it is getting and uses a recognizer tailored specifically for this kind of input. Another possibility is to send the input to all three recognizers and use confidence levels returned by the recognizers to choose the recognized result.

4.2.2. Grammar

Imposing a grammar on a sentence recognizer allows improved recognition accuracy. The recognition task in the InfoPad system involves word recognition and not sentence recognition, so using a grammar may not improve performance.

4.2.3. Constrained Writing Area

If the writing area were constrained, for example with ruled horizontal lines and tick marks, or with boxes, the recognizer would be able to segment the characters and estimate

their size more easily, aiding the recognition process. Constraining the writing area also makes users write more neatly and allows corrections to be done locally on a character-by-character basis rather than by re-writing the entire word.

4.2.4. Spatial Locality

The fact that handwritten input contains absolute spatial information means that context-dependent recognition can take place. By noting that input comes from different windows or from different regions in the same window the recognizer can determine the kind of recognition service required and also which application should receive recognized text.

One example application of this feature is in correcting misrecognized handwriting. The user writes on an entry widget and electronic ink provides user feedback. Then the ink is erased and replaced by recognized text. If the user writes on top of any recognized character, the captured writing can be interpreted as a single character that replaces the recognized character beneath it. Alternatively, the new input can be interpreted as a gesture and sent to a gesture recognizer.

Another example is the situation where there is a page for handwritten cursive entry and a single entry widget for entry of a file name via handprinting, both on the same screen. Using spatial locality, the recognizer can tell which kind of handwriting to expect. It can use a more task-specific algorithm and thereby achieve better accuracy.

4.2.5. Digitizer Resolution

The Logitech Gazelle digitizer used in the InfoPad system generates about 100 points/sec at a resolution of about 200 points/inch, although it is capable of 377 points/sec at 414 lines/inch. It is commonly accepted that in order to capture the details of normal writing, at least 200 points/inch and 100 points/sec are required [Tapp90]. However there is evidence that these requirements are too tight. Some commercial PDAs such as the Newton MessagePad successfully use digitizers with much lower spatial and temporal resolution. As handwriting recognition technology improves the need for high resolution digitizers will diminish. The lower limit of digitizer resolution is the screen resolution. This is true since user feedback is at screen resolution.

4.2.6. Available Computational Power

Increasing the computational resources available to run the handwriting recognizer allows the use of more sophisticated algorithms. For example, multi-algorithm recognition with voting can be used. We can also do less pruning, thereby reducing the probability of eliminating the correct answer early in the recognition process. With greater computational resources the recognizer will interfere less with other applications running on the same processor and will return its results sooner, improving user response times.

4.3. Recognition Requirements of the InfoPad

In the InfoPad system, handwriting recognition is used to specify file names, e-mail addresses and Universal Resource Locators (URLs) for the World Wide Web (WWW). It is not primarily intended for mass text entry. Recognition of drawn gestures or geometrical objects are also interesting capabilities that can be useful for the InfoPad.

In the sections below, we describe the handwriting recognition requirements of the InfoPad system and the kinds of recognizers needed to support these capabilities.

4.3.1. Non-Dictionary Words

File names, e-mail addresses and URLs are examples of non-dictionary words. It is not possible to limit the search space by specifying a finite vocabulary which contains all legal words within this set. Also, grammar may not be used to improve recognition accuracy. A character-based recognizer rather than a word-based recognizer is therefore required. This recognizer must take the form of a handprint recognizer rather than a cursive recognizer since cursive recognition requires a vocabulary.

4.3.2. Mass Text Entry

Since the InfoPad is not meant primarily for mass text entry, it is not essential to support this type of data entry. Nevertheless the InfoPad can support mass text entry if it had access to cursive handwriting recognition. Cursive recognition requires a vocabulary and would benefit from a grammar. A cursive recognizer would also be useful in cases where the user wants to send e-mail as recognized text rather than electronic ink.

4.3.3. Gestures

Gesture recognition has proven to be very useful in the Windows for Pen, PenPoint, and Newton OS environments. They are useful for short commands for applications as well as for correcting and annotating handwritten documents.

A gesture recognizer may be implemented as an independent recognizer or as part of a hand-print recognizer. Due to spatial locality, it is often possible for applications to determine whether handwritten input is meant to be a gesture or a printed character, so it is preferable to have a separate gesture recognizer.

4.3.4. Geometric Shapes

Many applications, including the circuit schematic recognizer described in Chapter 6, require recognition of drawn shapes. Other such applications include computer aided design of various kinds, including architectural drawings and clothing. To support these applications, a geometric object recognizer would be very useful. However, it would need to be highly customizable so that applications can easily specify the primitives to be recognized.

Alternatively, a library of recognition routines could be provided and applications can use these routines to create their own geometric object recognition engines.

4.4. Models for Providing Recognition Services

In this section, we discuss models for each facet of providing handwriting recognition services. They are the user interaction model, the programming model, and the service provision model.

4.4.1. User Interaction Model

The user interaction model describes the modes of interaction between the user and the handwriting recognizer. There are several alternatives. One model is for the user to write on a single space on the screen, with recognized text going automatically to the correct application. Another is for all handwritten input to be local to the application. That is, all characters are written in writing areas that belong explicitly to individual applications. The user interaction model also covers correction mechanisms. One model is for the user to per-

form corrections in a separate mode, while another model has the user write on top of the mis-recognized character. The recognizer detects the location of the new input and behaves accordingly.

4.4.2. Programming Model

The programming model defines the interface into the handwriting recognizer from the application programmer's perspective. There is a range of choices available in selecting the tightness of the coupling between the recognizer and the application. For example, some applications may not want to be aware of recognition at all, preferring to have the recognizer run separately and emulate a keyboard. Other applications may record pen strokes themselves, calling the recognizer using their own application-specific parameters. In this section, we explore some of these choices and their consequences for implementation.

4.4.2.1. Uncoupled Applications

Some applications, especially existing applications, should not need to be cognizant of the handwriting recognizer. They receive user input as though from a keyboard and mouse. Such applications can be supported by a stand-alone widget application in which the user writes on a special writing canvas. On recognition, this widget sends recognized text to the application with current recognition focus via the windowing system. Handwriting recognition focus may be determined by keeping track of the most recently active window or by other possibly more explicit means.

One disadvantage of this approach is that the application cannot tailor the recognizer to its own needs, so it may be necessary to give the widget separate modes for each application. This is cumbersome since the widget must be augmented for each additional application supported, unless the application comes with its own customized widget. However, this situation is not likely to arise often since all the kinds of applications expected on the InfoPad require the same print recognition service.

Another disadvantage of this approach is that extra screen area, always precious on a small portable terminal, is consumed. This is especially serious if separate widgets are needed for each application.

One major advantage of this approach is that the stand-alone widget can run on any remote machine using the X-Window system as a display mechanism, thereby supporting distributed processing. Another advantage is that the recognizer can be transparently replaced with a better one without modifying any of the applications.

4.4.2.2. Loosely Coupled Applications

An application may prefer a more customizable interface, where the programmer can design all user interaction himself. This can be supported by providing a set of widgets which the programmer uses in place of regular entry widgets within his application. Only minimal changes to his code are needed. He can also control some of the recognizer's parameters as necessary. However, he may not have access to all of the recognizer's internal state and functionality since the widget encapsulates the recognizer's details.

4.4.2.3. Tightly Coupled Applications

In some situations, an application may prefer to have complete control of the recognizer's parameters, including capturing ink itself and sending the ink to the recognition engine. The ink may represent print, cursive, gestures or drawn items. It is up to the application to decide which type of recognition is required and to access the recognizer with the appropriate parameters to perform the job. A comprehensive API into the recognizer is required, possibly including access to recognition confidence levels, the N best candidates, and vocabulary words.

4.4.3. Service Provision Model

The service provision model describes how and where service is provided. Service may be provided by a separate process running locally (or equivalently a separate thread), as a separate process running remotely, or as part of the same process.

As described in Section 3.5.3. on page 42, it is possible to run the recognizer remotely and to communicate with it using Internet sockets. This model of providing service is very powerful. It allows use of only one recognition engine per user rather than one recognizer per application. This engine can be time-shared between applications and even between

users. The recognition server may be upgraded transparently to provide the best service, and may be implemented on general purpose or special hardware.

Remote execution is the best service model in most cases, but there are some situations where it is worthwhile to compile the recognizer into the application. Such a situation arises when the recognizer is tightly coupled with the application and where the recognizer should not be shared, such as in batch recognition of pre-recorded speech, or where network access is expensive, or where the recognizer is highly customized for the application.

4.5. Properties of Printed Handwriting

In recognizing printed handwriting, we have to take account of several of its unique properties. When dotting i's and j's, and when crossing t's, writers often delay the stroke until after the entire word has been written. This delayed stroke is therefore out of sequence in the input stream and the recognizer must take care of this. This is especially common in cursive handwriting. One solution for print recognition is to sort all the handwritten strokes from left to right. Another is to look for strokes that are clearly out of sequence (based on tick marks or boxes in the writing area) and move these strokes backwards in the sequence to immediately follow the other strokes in the same box. The difference is that in one case, all strokes are sorted whereas in the other case only delayed strokes are used.

There are several characters which have similar uppercase and lowercase representations; the only difference is size. The recognizer must therefore have a mechanism to determine which case the writer is using. A commonly used solution is to constrain the user to write between lines and to use the line pitch as a hint. A related problem is that some characters such as g and j have descenders, so the lower baseline upon which the user writes must be above the lower boundary of the window to allow space for descenders.

Many characters have more than one written representation. For example, the letter r can be written in two ways (see Chapter 8). Even for the same representation of a particular character, there may be several ways of ordering or even writing the strokes. For example, in E the four strokes may be written in any order. In f, the vertical stroke may be written top to bottom or bottom to top.

Many users write words using characters that overlap their neighbors. Others write in a “run-on” fashion where they do not pick the pen up between characters. The recognizer therefore cannot easily use x coordinate separation to segment characters. One solution is to use an algorithm that does not require pre-segmentation. Another is to use boxes or tick marks along the lower baseline.

There are a few characters which cannot be differentiated without context. The numeral 0 and the letter O are written identically, and so are the numeral 1 and the letter l. The application must therefore tell the recognizer the context so that it can differentiate between these characters.

Variation between writers can take several forms. Left handed writers sometimes write strokes in the opposite direction to that of right handed writers. For example, a right handed writer crosses a t from left to right whereas a left handed writer crosses it from right to left. Some writers slant their characters to the left, some to the right, and some not at all. Some do not have a consistent slant. Writing speed varies among writers as well.

Visual feedback affects the quality of printed handwriting. If the application does not generate electronic ink to follow the pen tip on the screen, the user is more sloppy. Writing on lines or in boxes also usually results in neater handwriting.

4.6. HMM Based Handwriting Recognition

A handwriting recognition algorithm must take into account the characteristics of handwriting described in the previous section. As illustrated in Figure 4-1, HMM based recognition is done in several steps. Some of these steps may be omitted in some implementations. The first step is preprocessing raw handwriting. This includes interpolating, sub-sampling, and normalizing the raw data to eliminate dependence on writing speed and digitizer resolution.

Segmentation is done next based on information from the geometry of the writing area. It is often done in several alternative ways. Each alternative segmentation is passed to the next stage for further processing and eventual elimination of incorrect segmentations. Fea-

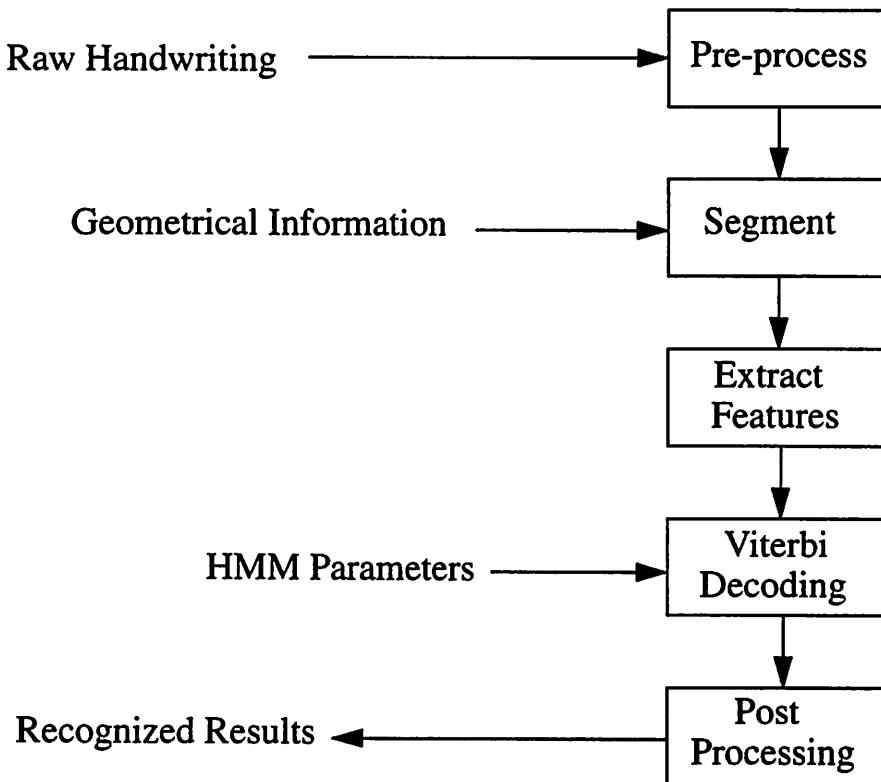


Figure 4-1. High-Level Description of a HMM Based Handwriting Recognizer

tures may then be extracted and passed to the Viterbi decoding engine, which uses HMM parameters to generate recognition results for each segmentation.

Post processing involves choosing the best answer from the alternatives provided by the Viterbi decoder, including disambiguating uppercase and lowercase characters.

4.6.1. Hidden Markov Modeling

In this section, we describe the hidden Markov model in general terms. This modeling technique is used in the handwriting recognizer described in Section 4.7 and in both the speech recognizers described in Chapter 5.

Hidden Markov modeling is used to model a statistical process which has a finite number of states. The process transitions from one state to another at each cycle; the destination (or successor) state is determined probabilistically and depends on the current state only. It does not depend on the states before the current state. That is, the probabilistic state process

is a Markov process and each state transition is independent of the previous state transition. Each state generates an output from a finite set of observations; the actual output produced depends probabilistically on the current state. In some applications of the hidden Markov model, the output depends on both the current and successor states. But in the recognizers described in this thesis the output depends only on the current state. The model is called “hidden” because we can observe only the outputs of the Markov process and not the states themselves.

The model has many parameters. For each transition between states, the transition probability:

Equation 4-1.

$$A_{ij} = P(s_{t+1} = j | s_t = i)$$

is the probability that the successor state to state i is state j . There is therefore a transition probability associated with every pair of states. The current state is a valid successor to itself if the process can stay in the same state for more than one cycle. This allows implicit time-duration modeling.

The output probability:

Equation 4-2.

$$B_i(O_t) = P(O = O_t | s_t = i)$$

is the probability that the observed output O_t is generated if the process is currently in state i . There is therefore an output probability distribution associated with every state, and each distribution has a probability value for every valid output observation value. In some systems, this output can be continuous rather than discrete, which results in probability distributions that are continuous.

The probability that any given sequence of observations $O = \{O_1 \dots O_T\}$ matches any sequence of states $S = \{s_1 \dots s_T\}$ for T cycles is:

Equation 4-3.

$$P(O|S) = B(O_1) \prod_{t=2}^T A_{s_{t-1}, s_t} B_{s_t}(O_t)$$

Equation 4-3 can be used to determine the most probable state sequence via Bayes’ Rule:

Equation 4-4.

$$P(S|O) = \frac{P(O|S)P(S)}{P(O)}$$

Since we are looking for $\text{MAX}(P(S|O))$ over all state sequences S and $P(O)$ is independent of S , the latter term is irrelevant to the evaluation of the MAX operation and may be ignored. Therefore, maximizing Equation 4-3 is equivalent to maximizing Equation 4-5:

Equation 4-5.

$$P = \pi(s_1)B(O_1) \prod_{t=1}^{T-1} A_{s_t s_{t+1}} B_{s_{t+1}}(O_{t+1})$$

where $\pi(s_1)$ encapsulates $P(S)$ and is the a priori probability that the state s_1 is at the start of a sequence. This equation may be evaluated inductively using the forward algorithm:

Equation 4-6.

$$\alpha_1(i) = \pi(i)B_i(O_1)$$

Equation 4-7.

$$\alpha_t(j) = \left[\sum_i \alpha_{t-1}(i)A_{ij} \right] B_j(O_t)$$

Equation 4-8.

$$P = \alpha_T(s_F)$$

where s_F is the final state in the sequence S . The forward algorithm gives us the probability of that state sequence, which is adequate for determining the most probable single character. But for word recognition, this gives the probability that s_F is the final state in the sequence but does not allow us to trace back the path through the sequence and thereby determine the most probable character sequence. We therefore make the Viterbi assumption:

Equation 4-9.

$$v_t(j) = [\text{MAX}(v_{t-1}A_{ij})]B_j(O_t)$$

to replace Equation 4-7. Since we are taking the most likely path, we are able to store pointers back along this path and trace through it after the final state is processed to determine the most likely character sequence.

The transition probabilities $\{A_{ij}\}$ and the output probability density functions $\{B_i\}$ are trained using the Baum-Welch re-estimation algorithm [Baum72].

For a more detailed explanation of hidden Markov models, refer to [Juan84].

4.7. A Writer Independent Handprint Recognizer

A writer independent hidden Markov model based handprint recognizer was developed and deployed in the InfoPad system. It was inspired by the author's experience with HMM-based speech recognition [Rabaey88]. Two sets of Markov model parameters were trained, one for 61 alphabetic and special characters and another for all 10 digits.

In this section, we discuss the recognition algorithm, the character sets selected, the applications programming interfaces, the software written for capturing and manipulating handwritten data, and the training data set. Except for the graphical user interfaces which were written in Tcl/Tk [Oust94], all the software was written in C and C++.

4.7.1. The Recognition Algorithm

The recognition algorithm begins with preprocessing and feature extraction, then does a Viterbi search, and then traces back through the HMM trellis to determine the most likely character written. Some post-processing is done to disambiguate uppercase and lowercase characters.

4.7.1.1. Heuristics

In order to improve recognition accuracy, heuristics are applied to the input pixel sequence before preprocessing. These heuristics detect certain characters, which are listed in

Character	Heuristic
. (period)	One stroke, less than 5 pixels
: (colon)	Two strokes, less than 5 pixels each

Table 4-1. Heuristics Used in Handwriting Recognition

Table 4-1. These characters are otherwise difficult to detect due to the small number of pixels and the algorithm's scaling behavior described in the next section.

4.7.1.2. Preprocessing and Feature Extraction

Several processing steps are performed to extract features before performing the actual Viterbi search on the Markov model. These steps are called preprocessing and consist of truncation, sorting, segmentation, normalization and feature extraction. The strokes are not

sub-sampled spatially or temporally because experiments with sub-sampling showed that it reduces recognition accuracy, especially for characters with a small number of points per stroke.

As soon as the recognizer receives a sequence of strokes it computes the centroid of each stroke. The strokes are sorted left-to-right and then segmented into characters. Segmentation is done by assigning each stroke to a character using the position of its centroid with respect to the tick marks provided on the drawing canvas. The last 3 pixels are then truncated from each stroke that has 15 or more pixels. This helps reduce the effects of the artifact that arises on pen-up, as described in Section 6.1.3. on page 116.

The characters are then processed individually. All strokes within each character are normalized vertically based on the highest and lowest points of all the strokes in the character. The recognizer quantizes the y-coordinates to 256 levels.

Lastly, the feature triplets are calculated for each pixel. In this algorithm, very simple geometric features are used: slope, slope difference and y-coordinate for each pixel, each quantized to 8 bits (256 levels). The sequence of feature triples is calculated for each stroke in the character and concatenated with the previous strokes' features. This concatenated array of feature triplets is passed on to the Viterbi search engine.

4.7.1.3. The Hidden Markov Model

Before describing the Viterbi engine used in this recognizer, we describe the Markov models used in this recognizer. Each character is assigned a number of Markov states based on its complexity. The number of states is listed in Table 4-2 for the 61-character recog-

Character	States	Character	States	Character	States
a	8	A	12	* (asterisk)	10
b	8	B	16	@ (at sign)	12
c	6	C	6	! (exclamation mark)	6
d	12	D	10	- (minus sign)	2
e	10	E	10	. (period)	1

Table 4-2. Number of Markov States for Each Handwritten Character in the 61-Character Recognizer

Character	States	Character	States	Character	States
f	8	F	8	/ (slash)	4
g	10	G	8	~ (tilde)	4
h	10	H	10	tt (double t)	10
i	5	I	10	_ (underscore)	2
j	8	J	10		
k	10	K	10		
l	4	L	6		
m	16	M	16		
n	10	N	12		
o	8	O	8		
p	10	P	10		
q	12	Q	10		
r	8	R	16		
s	8	S	8		
t	8	T	8		
u	10	U	10		
v	8	V	8		
w	14	W	14		
x	8	X	8		
y	8	Y	8		
z	10	Z	10		

Table 4-2. Number of Markov States for Each Handwritten Character in the 61-Character Recognizer

nizer and in Table 4-3 for the digit recognizer. The number of states for each digit or character was initially chosen based on the complexity of the character, and then refined using an iterative process to assign more states to characters with higher error rates.

Many of the characters can be written in several ways. In the 61-character recognizer all the variations of each character are condensed into one model whereas in the digit recognizer each distinct way of writing each digit has a unique model. Therefore there can be several entries for each digit in Table 4-3. The various forms of each character and digit are documented in Chapter 8 on page 138.

Digit	States	Digit	States	Digit	States
0a	8	5a	10	8a	10
1a	4	5b	10	8b	10
1b	8	5c	10	8c	10
1c	8	5d	10	8d	10
2a	7	6a	6	8e	10
2b	8	6b	10	8f	10
2c	8	6c	8	8g	10
3a	12	7a	6	9a	8
4a	8	7b	8	9b	10
4b	10	7c	10	9c	10
4c	12	7d	8		

Table 4-3. Number of Markov States for Each Handwritten Character in the Digit Recognizer

All characters models use left-to-right topologies as shown in Figure 4-2. Transitions may

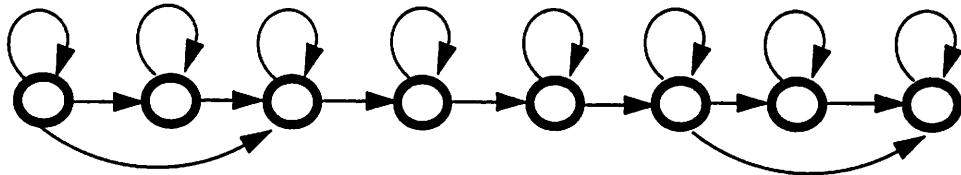


Figure 4-2. Topology of an 8-State Handwriting Markov Model

take place from any state to itself or to the next state only, except at the beginning and ending states. The beginning and ending states have extra transitions to compensate for artifacts on pen-down and pen-up. These artifacts are context dependent and may be modeled as co-articulation effects with adjacent characters.

The output probabilities are estimated using discrete probability distributions (discrete HMMs) which are trained using a counting approach called the Baum-Welch algorithm [Baum72]. This approach is iterative so that the model automatically aligns with the data.

4.7.1.4. Viterbi Algorithm

The recognizer uses the Viterbi algorithm to decode the Markov model. All parameters are represented in floating-point. The algorithm calculates Equation 4-5 for all characters in the vocabulary. The character with the highest probability at its final state is determined to be the recognized character.

This recognizer's HMM parameters are severely undertrained, as explained in Section 4.7.5. Some preprocessing of the trained parameters is therefore done in order to improve the models' generality and thereby improve recognition accuracy for characters that are not in the training set.

There are two parameter preprocessing steps in the algorithm. Each output probability is subjected to a floor of 0.0001 so that all output probabilities below this value are set to it. This prevents any state sequence from becoming extremely improbable. Without this step, a single mismatch due to the undertrained models could make the correct answer highly improbable.

Each output probability distribution is subjected to a 7-pixel filter window (3 pixels on each side of the current pixel). The probability of each feature value is set to the highest probability within this window. This helps smooth out the probability distribution function and thereby generalize the recognizer. Experiments showed that using the median rather than the highest probability within the window does not work as well. The sparseness of the training data meant that most of the probability values in the distributions are almost 0, so the median value is usually 0.

If the models were not undertrained, it would not have been necessary to take either step. There are other ways to deal with this problem. Many researchers deal with undertraining by fitting the probability distribution to a standard curve such as a Gaussian. This would have been the next step if the current recognition algorithm had not given sufficient recognition accuracy (see Section 4.7.6).

4.7.1.5. Post-Processing

Some characters have similar upper case and lower case representations. These characters are post-processed to determine the case by comparing their height to the height of the writing area. They are: c, m, n, o, p, s, u, v, w, x, y, and z. Once the case of each character has been determined, all the recognized characters in the word are concatenated into a string and this string is returned to the calling application.

4.7.2. Character Sets

The character sets for the recognizer were chosen based on the anticipated needs of InfoPad applications. For e-mail addresses, URLs for the World Wide Web and file names, the 61-character recognizer supports all 52 upper and lower case letters, at-sign “@”, period “.”, slash “/”, tilde “~”, and underscore “_”. The asterisk “*” allows wildcard characters in file selection boxes. The “double t” character models two consecutive “t”s which are crossed with a single bar.

For numeric entry of parameters into applications, a separate set of HMM parameters was trained. In the InfoPad model, multiple recognizers are easily accessible so applications that require digit recognition can connect directly to a digit recognizer regardless of whether they also need other recognition services. A dedicated digit recognizer achieves much better recognition accuracy than a single all-purpose recognizer.

Another advantage of a separate digit recognizer is that some easily confusable characters can be differentiated. For example, the numeral “0” and the letter “O” are easily confusable, as are the numeral “1” and the letter “l”.

4.7.3. Applications Programming Interfaces

There are currently three ways for applications to access the handwriting recognizer. A handwriting recognition widget runs as a separate process and allows uncoupled applications to accept handwritten input. A remote server allows applications to access a recognizer over the Internet and to control it remotely. A software library with a standard API allows applications to tightly couple with the recognizer for greater customization of the recognizer to the application.

4.7.3.1. Handwriting Recognition Widget

A screen shot of the handwriting recognition widget is shown in Figure 4-3. This widget

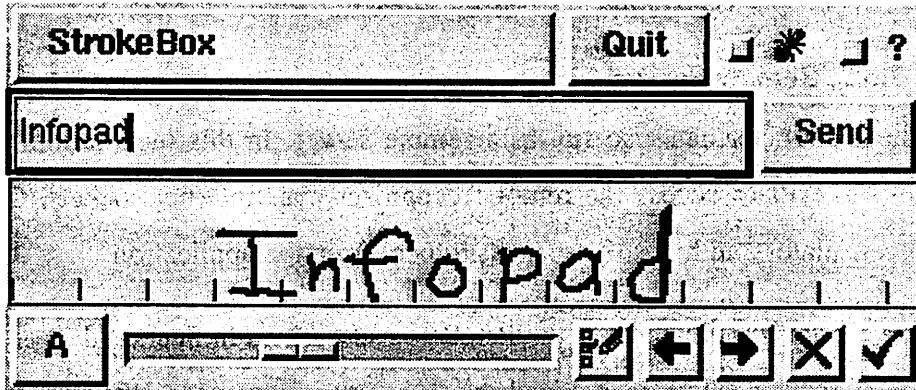


Figure 4-3. Screen Shot of the Handwriting Recognition Widget

was implemented by Armando Fox and uses the client side of the Sun API described in Section 4.7.3.3 to interface with the recognizer engine. It runs as a separate process, collecting handwritten input, running the input through the recognizer, and sending recognized text to the current application via the X server. The user writes in a box with tick-marks. Recognized results appear in a separate text window unless there is another currently registered application which should be receiving recognized text, in which case recognized text is sent directly to that application. A slider controls the pitch of the tick marks and other buttons allow various editing functions such as backspace, space bar, disable recognition, and recognize. The name of the current application is shown in the box on the top left corner; this is the application which receives recognized text.

The button on the lower left corner of the widget allows the user to switch between recognizers. When the letter “A” shows, the 61 character recognizer is the current recognizer. If the number “9” shows, the 61-character recognizer is dormant and the digit recognizer is active.

Currently, the widget works with applications which use the tk widget set [Oust94], in particular the tk entry widget. It modifies the class level bindings of all tk entry widgets on the current screen so that when there is a pen tap in that widget, a message is sent to the hand-

writing recognition widget to register the application as the current application so that it will receive recognized text.

4.7.3.2. Remote Server

The recognizer can also run as a remote server. In this incarnation, applications use a custom API to access the remote recognizer over Internet sockets. Of course, the API encapsulates and hides the socket protocol from the application.

The recognizer connects to the Pen Server described in Section 3.3. on page 32 to obtain pen data. This data is recognized and the results are passed to the application. The application does not need to capture pen pixels itself. This recognizer can be configured at start-up to use either the 61-character set or the digit set described in Section 4.7.2 above. Command line options allow the user to specify:

- the character set (vocabulary)
- a list of files of handwritten input for batch recognition (see Section 4.7.4 for data file format)
- the **floor** and **filter width** parameters (they default to 0.0001 and 3 respectively, as described in Section 4.7.1.4)
- the ratio for sub-sampling (the ratio is multiplied by the height of the character to determine the minimum allowable distance between pixels; intervening pixels are deleted)
- the number of pixels to truncate from the end of each stroke to allow for co-articulation (see Section 4.7.1.2 for default behavior)
- the host name and port number of the Pen Server from which to read a stream of pen-resolution data (unnecessary for batch mode operation)
- the Internet port number on which to accept connections from applications (defaults to 1510)

Refer to the manual pages for more details.

An application that wishes to use the remote handwriting recognition server must compile the client side API library into itself. Using the client API, the application opens a connection to the remote handwriting recognizer using `HW_OpenConnection()`:

```
recogConnection *HW_OpenConnection (char *host, int port)
```

The **recogConnection** data type is a structure that contains information about the current recognizer. **host** is the name of the remote machine on which the recognizer is running and **port** is the Internet port number on which the recognizer is accepting connections.

To begin recognition, the application issues **HW_SendRecognize()**:

```
void HW_SendRecognize(recogConnection *conn)
```

This command tells the recognizer to start recording handwritten input and to store it for recognition. When the application is ready to receive recognized text, it issues **HW_GetWord()**:

```
void HW_GetWord(recogConnection *conn, char *word)
```

This command tells the recognizer to perform recognition on the captured handwriting and return the results in the buffer **word**. If the application wants to suspend recognition, it may issue **HW_SendIdle()**:

```
void HW_SendIdle(recogConnection *conn)
```

This tells the recognizer to stop capturing handwriting and wait for further instructions. If the application wishes to disconnect from the recognizer, it may issue **HW_CloseConnection()**:

```
void HW_CloseConnection(recogConnection *conn)
```

This command tells the recognizer to disconnect from the application. The recognizer stays active and waits for another connection from the same or another application.

4.7.3.3. Sun API

The handwriting recognizer can be included into applications as a software library. In this incarnation of the recognizer, an API from Sun Microsystems [Kemp93] is used since this is the only available standard API for UNIX handwriting recognition engines. By adhering to this standard, the recognizer can be useful outside the InfoPad project and InfoPad applications can use recognizers from other institutions.

The Sun API is designed for applications which capture electronic ink and send it to the recognizer themselves rather than relying on the recognizer to obtain electronic ink via the Pen Server. Each application may allocate as many recognizers as it likes at run time via UNIX dynamic library loading. The client-side interface (used by applications) is not the same as the server-side interface (the engine's API) and the software distribution from Sun provides the code to map between the two interfaces. However there is a close correspondence between the two APIs. In the rest of this section we describe the server-side API. For details on the client-side API, refer to [Kemp93].

Applications must first initialize and allocate the recognizer by calling `__recognizer_internal_initialize()`:

```
recognizer __recognizer_internal_initialize(void* handle,  
                                             rec_info* ri)
```

This routine initializes and allocates memory for the recognizer indicated by `handle` using parameters specified in `ri`. All structures created and used by the recognizer are stored in the item of type `recognizer` which is returned by the initialization function. This allows the use of more than one recognizer at one time. A recognizer may be deallocated and deleted using `__recognizer_internal_finalize()`:

```
recognizer __recognizer_internal_finalize(recognizer rec)
```

Once initialized, the recognizer may be given recognition-related commands. To send strokes to the recognizer, call `set_buffer()`:

```
int set_buffer (recognizer r, u_int nstrokes,  
                pen_stroke *stroke)
```

This routine appends `nstrokes` strokes to the recognizer's internal stroke buffer. This internal buffer can be cleared by calling `clear()`:

```
int clear (struct _Recognizer* r, bool delete_points_p)
```

When all relevant strokes have been added to the buffer, the recognizer is instructed to perform recognition using `translate()`:

```
int translate (struct _Recognizer* r, u_int nstrokes,
```

```
pen_stroke* strokes, bool correlate_p, int* nret,  
rec_alternative** ret)
```

This routine appends **nstrokes** strokes to the buffer and then performs recognition. The **correlate_p** flag tells the recognizer whether or not to correlate strokes with characters and return correlation information to the application. Correlation information allows applications to determine which strokes correspond to which character. **nret** is the number of alternative translations returned by the recognizer, and **ret** is an array of alternative translations. Each translation is a linked list of the characters which constitute the recognized word.

The geometry of the writing area is specified using **set_context()**:

```
int set_context (struct _Recognizer* r, rc* rec_xt)
```

rec_xt contains the baseline and tick mark positions and the boundaries of the writing area.

The API includes several other routines for ink manipulation and others for controlling the recognizer, some of which are not used.

4.7.4. Data Capture and Manipulation

A software package was written for handwriting capture and manipulation. This package is available both as a software library to be included into applications and as a stand-alone application with its own graphical user interface. The library version is incorporated into the recognizers described elsewhere in this section. The main menu bar of the graphical user interface of the stand-alone version is shown in Figure 4-4. The graphical user inter-



Figure 4-4. Main Menu Bar for the Data Capture and Manipulation Package

face was written using tcl and tk [Oust94] while the capture and manipulation routines were written in C and C++.

The **File** menu supports the usual file reading and writing commands. There are three kinds of files. Word files contain raw handwritten input as a sequence of pixels. The format of

such files is illustrated by the example in Figure 4-5. **Text** is the text represented by the

```
Text= pqrst_
Writer= shankar1
StrokeCount= 2
UpperBaseline= 4500
LowerBaseline= 5000
PixelCount= 6
4188 4867
4190 4867
4190 4871
4191 4896
4196 4944
4196 4964
PixelCount= 8
4211 5325
4212 5300
4213 5241
4213 5207
4209 5135
4205 5098
4198 5033
4196 5004
```

Figure 4-5. Example of Word File Format

data in the file. **Writer** indicates the person who wrote the text. **StrokeCount** is the number of strokes comprising the word or character. A stroke is defined as a sequence of pixels from pen down to pen up. **UpperBaseline** and **LowerBaseline** are the positions of the lines between which the user writes in the writing canvas. See Figure 4-8 below for a picture of the writing canvas.

PixelCount is the number of pixels in the current stroke. These pixels' x and y coordinates appear after **PixelCount**. The next and all subsequent strokes are appended after the current stroke.

Feature files contain the features extracted from the Word file. These features are described in Section 4.7.1.2. The syntax of a Feature file is illustrated in Figure 4-6. **Text**, **Writer** and **StrokeCount** are identical to the equivalent fields in a Word file. **xmax**, **ymax**, **xmin**, and **ymin** are the maximum and minimum x and y coordinates in the entire word. **PixelCount** is the number of pixels in the stroke that follows. **xmax**, **ymax**, **xmin**, and **ymin** are

```

Text= <Text>
Writer= <Writer>
StrokeCount= <number_of_strokes>
xmax= <xmax>
ymax= <ymax>
xmin= <xmin>
ymin= <ymin>
PixelCount= <number_of_pixels>
xmax= <xmax>
ymax= <ymax>
xmin= <xmin>
ymin= <ymin>
<slope_1> <slope_diff_1> <y_coord_1>
<slope_2> <slope_diff_2> <y_coord_2>
.
.
.
PixelCount= <number_of_pixels>
xmax= <xmax>
ymax= <ymax>
xmin= <xmin>
ymin= <ymin>
<slope_1> <slope_diff_1> <y_coord_1>
<slope_2> <slope_diff_2> <y_coord_2>
.
.
.

```

Figure 4-6. Feature File Syntax

the maximum and minimum x and y coordinates for that stroke, and they are followed by the feature triplets for each pixel. The next and subsequent strokes follow.

Vector files encapsulate all the extracted features in the format expected by the HMM recognition software. The syntax is illustrated in Figure 4-7. **frames** is the number of feature triplets in the file. **dimension** is the number of features per pixel, which is 3 in this case. **codeBooksize** is the number of values each feature can take. Since they are quantized to 8 bits, this number is 256. The feature triples then follow.

The **Macros** menu contains several complex commands which are ordered sets of other simpler commands that may be accessed via other buttons on the main menu bar. These macros include a data capture session where the user is prompted to enter a list of words to

```

frames= <number_of_frames>
dimension= 3
codeBooksize= 256
<slope_1> <slope_diff_1> <y_coord_1>
<slope_2> <slope_diff_2> <y_coord_2>
.
.
.

```

Figure 4-7. Vector File Syntax

be stored and used later for training or for testing the recognizer. Other macros allow the user to sequentially view all the files in a directory or to view files by user. A set of hand-written words may be segmented and labeled using yet another macro. Vector files may be generated in batch mode via a macro. Since tcl is an interpreted language, it is very easy to add new macros.

The **Capture** menu allows the user to choose the input device and immediately write a word of handwritten data. The input devices currently supported are the mouse and two models of Wacom digitizer. A window with baselines (see Figure 4-8) comes up to accept

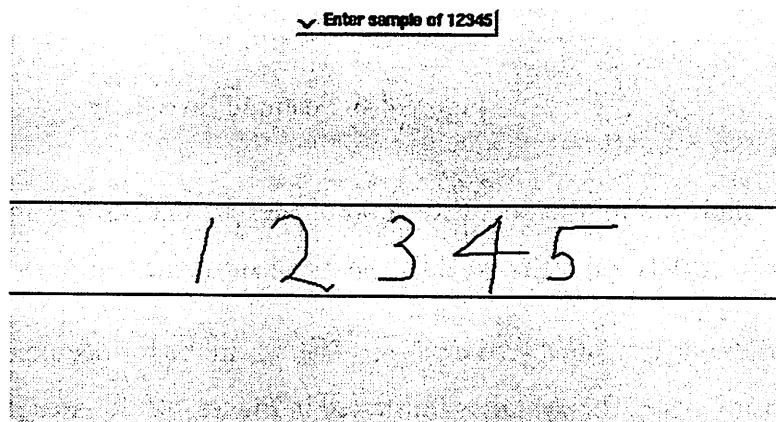


Figure 4-8. Handwriting Capture Canvas

user input.

The **Operations** menu allows the user to manipulate the current data set. Words may be truncated (the last few pixels are removed to alleviate the effect of the pen-up artifact), seg-

mented based on x-coordinate spacing, sorted (strokes are sorted left-to-right based on centroid), and sub-sampled. Feature extraction may also be done via this menu.

Word and feature files can be viewed via the **View** menu. Words on the canvas can also be queried to determine stroke order and to obtain other information about each stroke or word via this menu.

The **Train** button runs a batch training session on data stored on disk. The **Recognize** button tells the application to prompt the user to write a handwritten word on a pop-up canvas and to recognize this word using the current recognition parameters.

The **Globals** menu brings up a window showing the value of each global variable used by the application and to edit these globals. On start-up, the application also displays all the

global variables in a separate window, shown in Figure 4-9. **Word** is the currently dis-

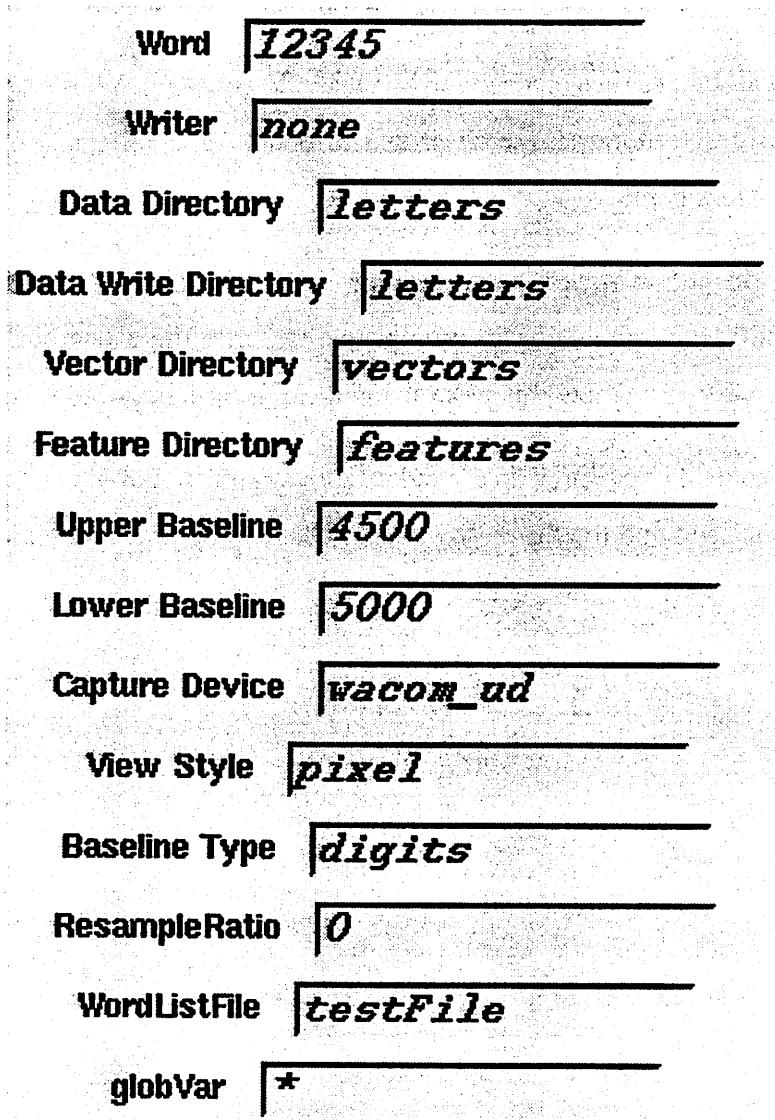


Figure 4-9. Global Variable Display/Editing Window

played or most recently written word. **Writer** is the current user. **Data Directory** is the path to the directory from which Word files should be read. **Data Write Directory** is the directory in which to save captured handwriting. **Vector Directory** and **Feature Directory** point to the locations where vector and feature files respectively are to be written. **Upper Baseline** and **Lower Baseline** indicate the positions of the lines between which the user writes. These parameters are in pen resolution coordinates unless the mouse was used as the capture device, in which case they are in screen resolution coordinates. **Capture**

Device indicates the current data entry device which can be either the mouse or a Wacom tablet. **View Style** indicates whether handwriting should be displayed as dots only or whether the dots should be connected. **Baseline Type** indicates normalization mode. This tells the feature extraction routines whether to normalize based on the baselines or the height of the characters. **Resample Ratio** indicates the minimum spacing between adjacent pixels when the input is sub-sampled. **WordListFile** tells the data capture macro where to find the list of words with which the user is to be prompted during a data capture session. **globVar** allows users to use wild cards when retrieving a large number of files for viewing.

The **Options** menu on the main menu bar allows the user to control display options. The **Misc** menu contains miscellaneous commands that were mainly used for testing and debugging while the **Help** button brings up a manual page describing the operation of the application.

4.7.5. Training Set

The recognizers were trained on very small data sets. Collecting a large data set requires a very large investment in time and effort. The 61-character recognizer was trained using 3626 captured characters from 16 writers. This corresponds to an average of 59 examples of each character. The digit recognizer was trained using 377 captured characters from 10 writers which is an average of 12 examples per model since there are 31 HMMs in this recognizer.

Training data was captured using the capture and manipulation package described in Section 4.7.4 above. Users were prompted to write words from two word lists. The raw captured data is stored in files on disk.

Segmentation was done automatically with operator intervention to correct segmentation errors and to attach a label to each character. First, the stroke sequence was sorted left-to-right to move delayed strokes to be adjacent to the main character. The segmentation macro looked for overlap in the x coordinate used this to cluster strokes into characters. Strokes were added or removed by the operator and a label was attached before segmentation was accepted. Segmented characters are stored one per file.

Another macro was used to read all the segmented characters from disk and generated feature vectors as described in Section 4.7.1.2. These vectors are also stored using one file per character. The training program then read the files one at a time to train the Markov model parameters. The syntax of the files used to store HMM parameters is shown in Figure 4-

```
HidMarkMod <HMM_name>

state <start_state> START
edge generic <start_state> <state_1> <start_state>_<state_1>
edge generic <start_state> <state_2> <start_state>_<state_2>
state <end_state> END
state <state_1> state <pdf_1_name>
edge generic <state_1> <state_1> <state_1>_<state_1>
edge generic <state_1> <state_2> <state_1>_<state_2>
state <state_2> state <pdf_1_name>
edge generic <state_2> <state_2> <state_2>_<state_2>
edge generic <state_2> <state_3> <state_2>_<state_3>
.
.
.

NoOutputPDF <start_state>_<state_1> generic <trans_prob_1>
.
.
.

OutputPDF <pdf_1_name> vector 256 numVec 3 map 0 1 2
{
<output_prob_1> <output_prob_2>... <output_prob_256>
}
OutputPDF <pdf_2_name> vector 256 numVec 3 map 0 1 2
{
<output_prob_1> <output_prob_2>... <output_prob_256>
}
.
```

Figure 4-10. HMM Parameter File Syntax

10. The first line specifies **HMM_name**, which is the name or label of the character modeled by this file. The section following it declares the states and the topology of the Markov model. **state** declares a state in the model and specifies whether it is a **START**, **END** or

normal state. Normal states have output probability distributions associated with them whereas the **START** and **END** states do not. **edge** declares an edge, or transition, between states and assigns a label to that transition representing the transition probability.

The next section specifies the transition probabilities between the states. **NoOutputPDF** declares that the edge itself does not have an output probability distribution associated with it, and assigns the transition probability for that edge. The last section uses the **OutputPDF** declaration to assign values to the output probability distributions which were declared in the **state** declaration.

4.7.6. Performance

Recognition accuracy was surprisingly good considering the small size of the training data sets. In user tests, the 61-character recognizer obtains about 80% character recognition accuracy on average, although there is wide variation among users. For the author, this recognizer achieves over 90% character accuracy. The digit recognizer achieves over 95% accuracy for all writers, with 100% accuracy for the author.

A test of the 61-character recognizer was conducted using a Wacom tablet on a Sun workstation. Each user wrote 27 words (115 characters) into the handwriting recognition widget described in Section 4.7.3.1. The results are tabulated in Table 4-4. The average recognition error rate is 17.0%.

We found that as users learned which characters were mis-recognized most often, they wrote those characters more carefully and got better results. The users therefore get trained to the system. We also found that some characters were similar enough to be easily confusable. Pairs of such characters are u-v, w-u, r-v, n-r, p-r, a-o and b-h. Usually, one character is recognized as the other but not vice versa. Some writers obtained mis-recognitions of particular characters consistently. However writing the mis-recognized character again more carefully usually solves this problem, and motivated users who played with the recognizer before the test did better. The long term solution is to train the system with more data.

Writer	Error Count	Error %
SN	10/115	8.7
NC	15/115	13.0
ML	21/116	18.1
TP	20/116	17.2
JR	30/115	26.1
IO	20/114	17.5
TB	49/115	42.6
SM	7/115	6.1
LG	10/115	8.7
YCC	18/115	15.7
JC	11/115	9.6
RS	32/115	27.8
HB	11/115	9.6

Table 4-4. Handwriting Recognition Results

4.7.7. Algorithmic and Implementational Improvements

There are several improvements which would benefit the handwriting recognizer. Currently, only one heuristic is used to improve recognition accuracy. Other heuristics can reduce the search space. The number of strokes in the character can be used to constrain the search. Descenders can be detected to reduce the search space. Delayed strokes are a sure sign that the character is either a t, i or j.

Output probability distributions should be fitted to Gaussians or other shapes to see if recognition accuracy can be improved this way. More training data should be captured and the models retrained. A large body of data with separate test and training sets is now available [Guyo94] for this task.

Other methods of estimating probabilities should be explored. Neural networks have been used by some researchers with good results (see Section 4.1.3 and Section 4.1.6). It is likely that a neural network based recognizer will perform better for probability estimation.

The recognizer currently returns only the most likely candidate. It should be enhanced to return the top few candidates together with a recognition confidence measure. Recognized results that are deemed highly uncertain (having low confidence) should be reported as

unrecognized. The recognizer should also detect spaces between characters so that users can write phrases.

A recognizer that recognizes both characters and digits at the same time should be trained up, so that the application does not have to direct the recognizer to use separate sets of parameters. This would be useful in applications which use single handwritten characters as command shortcuts and where alphanumeric input is required.

The version of the recognizer with the Sun API should be modified to execute remotely as a server. This allows the recognizer to run on a separate processor and therefore not load the current application's processor, and also gains all the other advantages of a remote server. The remote recognition server should accept and process partial input while the user is writing so that feedback is faster and overall latency is reduced.

The recognizer should also implement pruning to reduce the search space and thereby improve response times. Experiments need to be done to determine a good pruning threshold.

The current recognition performance figures are based on casual evaluation. A large test set is needed to test the system and benchmark its performance.

5 Speech Recognition

In this chapter we explore the speech recognition needs of the InfoPad system and describe the kinds of speech recognizers required to meet these needs. We then describe some models for delivering recognized speech and two different recognizers that were implemented, one of which was deployed in the InfoPad system. The other is a basis for a speech recognition server for InfoPad.

In the InfoPad system we do not use speech recognition to specify non-dictionary words since spelling them out verbally would be very time-inefficient compared to writing them. A spoken character recognizer could therefore play a secondary role, such as a fall-back to spell out misrecognized words. We look only at word recognition rather than character recognition, and, in any case, the former is a superset of the latter.

5.1. Performance of Existing Systems

In this section we review look at some systems reported in the speech recognition literature to motivate the design and deployment of the InfoPad recognizers. We look at a fully software solution from Carnegie Mellon University (CMU), then a solution from Bolt, Beranek and Newman (BBN) using special purpose off-the-shelf hardware, and then at a fully custom solution from a collaboration between the University of California at Berkeley (UCB) and Stanford Research Institute (SRI).

These solutions show that it is possible to get very high recognition rates on artificial tasks but that it is difficult to get real-time performance, especially for large vocabularies and for less constrained grammars. We can conclude that for small vocabularies it is possible to run a software speech recognizer in real time but for large vocabularies custom hardware may be required. As technology improves it becomes possible to run even large recogniz-

ers in software in real time. However custom hardware can help achieve a performance gain over a pure software system, and can allow use of algorithms which would otherwise be impractical due to their large computational requirements.

5.1.1. Software

The SPHINX speech recognizer [Lee89] was built at Carnegie-Mellon University (CMU) in the late 1980's and used hidden Markov modeling to represent speech. Several versions of the recognizer were built with different grammars. A bi-gram grammar gave SPHINX the best results, with accuracies of between 88.9% and 100% for a set of 15 speakers. The median recognition accuracy among these speakers was 96.6%.

The system had a 1000-word vocabulary, was speaker independent and worked on continuous speech. They used a statistical grammar of perplexity 60. This recognizer was state-of-the-art at the time. There is no data to indicate how long it took to run the algorithm, but it was almost certainly several times real time for that vocabulary.

5.1.2. Off-the-Shelf Hardware

A group at Bolt, Beranek and Newman (BBN) used an off-the-shelf board from Sky Computer that has an Intel i860 processor on board for speech recognition [Aust90]. They were able to run their algorithm in real time on this board, which was a factor of 5 speedup over straight C code on a SUN 4. The algorithm uses a fully connected first-order statistical grammar and has a vocabulary of 1,000 words. Recognition accuracy was 97.6%.

5.1.3. Custom Hardware

A collaboration between University of California at Berkeley (UCB) and Stanford Research Institute (SRI) resulted in a real-time 3,000 word speech recognizer built from custom hardware [Raba88]. The hardware ran the CMU and BBN algorithms above and also the DECIPHER system from SRI [Murv89].

5.1.4. Accuracy on Realistic Tasks

More recently, the performance of several research speech recognizers was measured in the DARPA Air Travel Information System (ATIS) common task domain [Pall92]. The test consisted of 971 utterances with 37 speakers, 17 male and 20 female. There were an

average of 11 words per utterance. The results are tabulated in Table 5-1. This test used

Institution	Word Error Rate (%)
AT&T	17.5
BBN	9.4
CMU	16.2
MIT	18.1
Paramax	10.6
SRI	11.0

Table 5-1. Accuracies for Recent Speech Recognizers

data collected from several sites, and is therefore more realistic than previous tests where all the data was collected at one site using one set of hardware. Also, some of the test data contained disfluencies. From the Table, we can conclude that word error rates of 10% to 15% are obtainable in realistic situations.

5.2. Characteristics of Speech Recognizers

There are several characteristics of speech recognizers that may be traded off against each other to produce the best performance in a given application. In this section, we examine these characteristics and the issues surrounding them, including how we might use them to improve recognition accuracy.

5.2.1. Word Length

Speech recognizers usually work better if the words in the vocabulary are longer. This is because the feature vectors for each utterance are longer so similarity measures have more data to work on. Longer words tend to be less confusable since the feature vector difference is greater on average. This assumes that there is sufficient training data. However, longer words tend to be used less often, which can lead to insufficient training data and under-trained models for those words. This can hurt recognition accuracy in some cases.

When there is enough training data, we can sometimes take advantage of the better recognition for long words. Compound words consisting of a concatenation of single words may be used instead of single words. For example, in the circuit schematic entry application described in Chapter 6 almost all commands are compound words, increasing average

word length and thereby increasing recognition accuracy. Again, this works best when the compound words are not undertrained.

5.2.2. Vocabulary Size

A larger vocabulary results in a larger number of confusable words. This means that recognition accuracy suffers. Of course, the converse is also true: a recognizer that uses a smaller vocabulary will generally perform better. In some situations, though, a smaller vocabulary does not help recognition accuracy. This happens when the words in the vocabulary are easily confusable.

We can take advantage of this feature by limiting the recognition vocabulary wherever possible, thereby increasing accuracy. However this requires that the recognizer vocabulary be configurable by the user or application programmer.

Although a larger vocabulary raises the average length of the words, this effect is not sufficient to overcome the effect of the larger number of confusable words so recognition accuracy often does decrease with increasing vocabulary size.

5.2.3. Grammar

Imposing a more constraining grammar on the recognizer improves recognition accuracy. This is because the space of allowable sentences becomes more constrained, allowing the system to eliminate illegal sentences or at least reduce their probability. However, grammar is often highly application-dependent. For example, a command-and-control application may prefer a different grammar than a dictation application. Obviously, an inappropriate grammar would increase recognition errors.

We can take advantage of increased accuracy from grammar by building a recognizer that allows user or application specific grammar. The recognition engine would be general enough to take a user-supplied grammar and apply it to the recognized words. This grammar may be a statistical or natural-language grammar, or some other kind of rule-base. A majority of the current systems use an n-gram grammar, where the probability of a word following another word is a function of its n predecessors.

5.2.4. Available Computational Power

Increasing the available computational power to run speech recognition allows us to use more sophisticated algorithms. For example, multi-algorithm recognition with voting could be used. We can also do less pruning, thereby reducing the probability of eliminating the correct answer early in the recognition process. With greater computational resources the recognizer will interfere less with other applications running on the same processor and will return its results sooner, improving user response times.

5.3. Recognition Requirements of the InfoPad

In the InfoPad system, speech recognition may be used to issue commands that drive applications or for dictation. In each of these two cases, the type of recognizer required and therefore the computational requirements are very different. As explained in Chapter 3, speech recognition is not efficient for specifying file names, entering other non-dictionary words, nor navigating the pointer across the screen. Therefore we do not address the requirements for providing these latter capabilities.

5.3.1. Commands

For commands, high recognition accuracy is essential. For any application, only a finite set of commands is valid at any one time, which means that only this finite set of commands needs to be in the vocabulary at that time. This set of commands can change dynamically as the state of the application changes. Therefore, a small recognition vocabulary with a simple rule based grammar is sufficient.

Recognition vocabulary and grammar for commands are specific to the application, and may change with the state of the application. Therefore, the application programmer must have the ability to specify the vocabulary and grammar for his application, independent of other applications. It is also desirable that the recognizer be flexible enough to adapt to changes in the state of the application. If the recognizer is able to keep track of the application's state, and can dynamically modify the vocabulary and grammar to depend on this state, it will be able to reduce the vocabulary and further constrain the grammar, thereby improving recognition accuracy.

Commands require high recognition accuracy because in the event of a mis-recognized command, the user may not have an opportunity to correct the error. Commands are executed immediately upon recognition and are especially dangerous for commands that erase or modify data where the application is unable to undo changes or erasures. The recognizer should use confidence measures to reject utterances which have interpretations that are not highly probable rather than execute a command that has low recognition confidence.

Because of the small vocabulary and simple grammar associated with the command recognition task, it is possible to create a small, fast recognizer for this application. Extra CPU power can be devoted to using a better algorithm since the search space has been reduced and the processing power required is therefore less than that required for large vocabulary recognition.

5.3.2. Dictation

The speech recognition requirements for dictation applications are very different from the requirements for command entry. Dictation requires a very large vocabulary and a generalized grammar, which is usually not highly constrained. Recognition of dictated speech is therefore a more computationally intensive problem than command recognition. However, recognition errors are more tolerable. A word-processing application will display the mis-recognized word on the screen, allowing the user to correct the error on the spot. However, in cases of automatic batch-mode transcription of recorded speech there is no operator to correct errors so there is a low tolerance of recognition errors. The key is interactivity. For interactive applications, a higher error rate is tolerable.

Dictation vocabularies and grammars tend to be standard, so the application programmer may not need to direct the recognizer at all. However there are some special cases where the application programmer or the user may want to add vocabulary words for a specialized domain, such as medical applications. For example, if a user is creating a document using dictation into FrameMaker, he may want to add domain-specific words to the general recognizer independently from FrameMaker. In this case, the recognizer must be controllable separately from the application so that the user can add vocabulary words independently of the application.

5.4. Models for Delivering Recognized Speech

There are several models that must be considered when providing speech recognition services. They are the user interaction model, the programming model, and the service provision model.

5.4.1. User Interaction Model

The user interaction model tells us how the user interacts with the recognizer. There are several possibilities. Some of these are described in the section on speech recognition focus below, but in the InfoPad system each application has to define its own user interaction model since there is no system level imposition of a model. The model also describes the correction mechanism in the case of recognition errors. This issue is addressed in Section 5.3 above.

5.4.1.1. Speech Recognition Focus

Speech recognition focus determines which application or applications receives recognized speech at a particular instant. If several applications are currently speech-enabled, it is often not obvious which application the user is talking to. For example, the user may be dictating into a word processor when the telephone application rings and he answers the call. The user then speaks to the telephone application and not the word processor, but the word processor does not automatically stop listening even if the pen focus moves to the telephone application. The user must explicitly tell the word processor's speech recognizer to stop listening.

There are several models for audio or speech recognition focus, and this issue is explored in greater detail when we talk about future work in Section 7.2.1.1. on page 133. Currently, the Audio Server does not provide a facility for applications, including recognizers, to determine who has audio focus. The speech recognizer's control widget allows the user some control of when the recognizer is active but there is no support for the applications programmer to determine who should receive audio or speech recognition focus, and there is no policy on audio focus.

5.4.2. Programming Model

The programming model defines the interface into the recognizer from the application programmer's perspective. There is a range of choices available in selecting the tightness of the coupling between the recognizer and the application. For example, some applications may not want to be aware of recognition at all, preferring to have the recognizer run separately and emulate a keyboard whereas other applications may record speech themselves, calling the recognizer using their own application-specific parameters, grammar and vocabulary. In this section, we explore some of these choices and their consequences for implementation.

5.4.2.1. Uncoupled Applications

Some applications, especially existing applications, should run independently without modifications for recognition. These applications can be supported by a stand-alone recognizer that sends recognized text to the application via a windowing system such as X or Windows. This is the approach taken by the DragonDictate commercial speech recognition system [Drag94].

In some cases the recognizer needs to be tailored to the application but not the other way around. The recognizer therefore has to know which application currently has speech recognition focus, and may make this determination by querying the windowing system to determine which application has current pointer focus. The issue of speech recognition focus is discussed in greater detail in Section 5.4.1.1.

5.4.2.2. Loosely Coupled Applications

Some applications may prefer a simple interface, using minimal recognition capability. Most applications would fall under this class. Their main functionality would be independent of recognition, and they can function without it. Such applications would control some parameters of the recognizer such as grammar and vocabulary, but the application software would not need major modifications to add recognition capability.

In this case, the recognizer could be compiled into the application or run as a remote process. The model for provision of recognition service is studied in greater detail in Section 5.4.3.

5.4.2.3. Tightly Coupled Applications

A few applications would benefit from being tightly coupled to the recognizer. For example, a dictation system which allows spoken correction and which stores audio for voice-mail, dictation or delayed recognition would want to capture its own audio data and send it to the recognizer, retrieving detailed results including a list of top recognition choices, confidence levels and temporal information. It may also want to control more recognition parameters than a loosely coupled application, or even perform its own segmentation and grammar processing.

In this case, the recognizer could be compiled into the application rather than run as a remote process. The issue of remote execution is examined in greater detail in Section 5.4.3.

5.4.3. Service Provision Model

The service provision model describes the way service is provided, regardless of the user interaction and programming models. Service may be provided by a separate process running locally, as a separate process running remotely, or as part of the same process.

As described in Section 3.5.3. on page 42, it is possible to run the recognizer remotely and communicate using Internet sockets. This model of providing service is very powerful. It allows use of only one recognition engine per user rather than one recognizer per application. This engine can be time-shared between applications and even between users. This recognition server may be upgraded transparently to provide the best service, and may be transparently implemented on general purpose or special hardware.

Remote execution is the best service model in most cases, but there are some situations where it is worthwhile to compile the recognizer into the application. Such a situation arises when the recognizer is tightly coupled with the application and where the recognizer

should not be shared, such as in batch recognition of pre-recorded speech, where network access is expensive, or where the recognizer is highly customized for the application.

5.5. A Small, Flexible Recognizer

A small, flexible recognizer was built by Andrew Burstein here at Berkeley [Burs96]. This recognizer was written in C++ and Tcl and is accessed via Tcl. It is phoneme-based and speaker independent. Several Tk widgets are provided for control of recording and recognition functions. Speech recognition functionality can be provided to applications with only a few lines of Tcl code. The vocabulary and grammar may be modified dynamically by the application.

5.5.1. Motivation

The objective of the speech recognizer is to provide InfoPad applications with a mechanism to receive commands and simple data by recognizing user speech. For example, the recognizer allows commands that are usually issued by selecting menu items, clicking buttons, or pressing function keys to be executed by speaking to the pad. Just as the names and contents of menus and button bars change from application to application, so too do the vocabulary and grammar. The speech recognizer accepts continuous speech (no pauses are required between words), and is speaker independent, but is capable of some adaptation to individual speakers. In its current form, it is intended for small to medium sized vocabulary applications such as command and control operation, but not general speech to text transcription.

The speech recognizer was designed as a compromise between the need to couple the recognizer tightly to the individual applications in order to increase recognition accuracy and the desire to make it as simple as possible for programmers to use speech recognition. Unfortunately, it is difficult to design a speech recognizer that can accurately recognize arbitrary English sentences, especially if speech is continuous. However, we can often narrow the recognition to a particular subject, such as commands one might give to a World Wide Web browser. This allows the recognizer to greatly reduce the size of the vocabulary and the complexity of the grammar that it must consider, thereby enhancing accuracy. Thus, it is important for the applications to tell the speech recognizer in advance

the vocabulary and grammar that it expects to receive from the user. This is a fairly simple task if the program is receiving verbal command and control: programs already have vocabularies and grammars defined by their button names, menu hierarchies, and typed commands.

5.5.2. Implementation

Programmers know what commands their programs can receive; the challenge is to make it easy for them to pass this information to and from the speech recognizer without forcing them to understand the details of speech recognition. We meet this challenge by implementing the speech recognizer as an extension to the Tcl language, by providing high level Tcl/Tk widgets and itcl objects for applications to interface with the recognizer, and by providing a graphical vocabulary and grammar editor.

All interaction with the speech recognizer can be handled through two Tcl commands, one controlling the recording of sentences and the other the actual recognition. Since the continuous-word recognizer recognizes entire sentences, not just words, the recorder supports automatic silence detection to locate the beginning and end of sentences. To facilitate error recovery, the recognizer provides a list of several top matches; thus, if the best estimate of the spoken sentence was not correct, the user can make a correction from the other choices. The recognizer is capable of determining if the user speaks a word that is not in its vocabulary rather than simply making the closest, albeit poor, match.

While the Tcl commands allow complete, low-level control over recording and recognizing speech, the most common tasks are controlled by the programmer through high-level itcl objects and by the user through itcl mega-widgets. These objects contain procedures to allow the programmer to manipulate vocabularies and grammars and to perform recognition without having to deal with the underlying data structures. These procedures also enable the programmer to convert spoken words into their phonetic transcriptions, so that new words can be added to the vocabulary as they are spoken by users.

A similar mechanism allows users to train the recognizer to adapt to *their own* particular pronunciation of any given word, if it differs significantly from the “standard,” speaker-independent pronunciation. The mega-widgets are designed to be dropped into programs

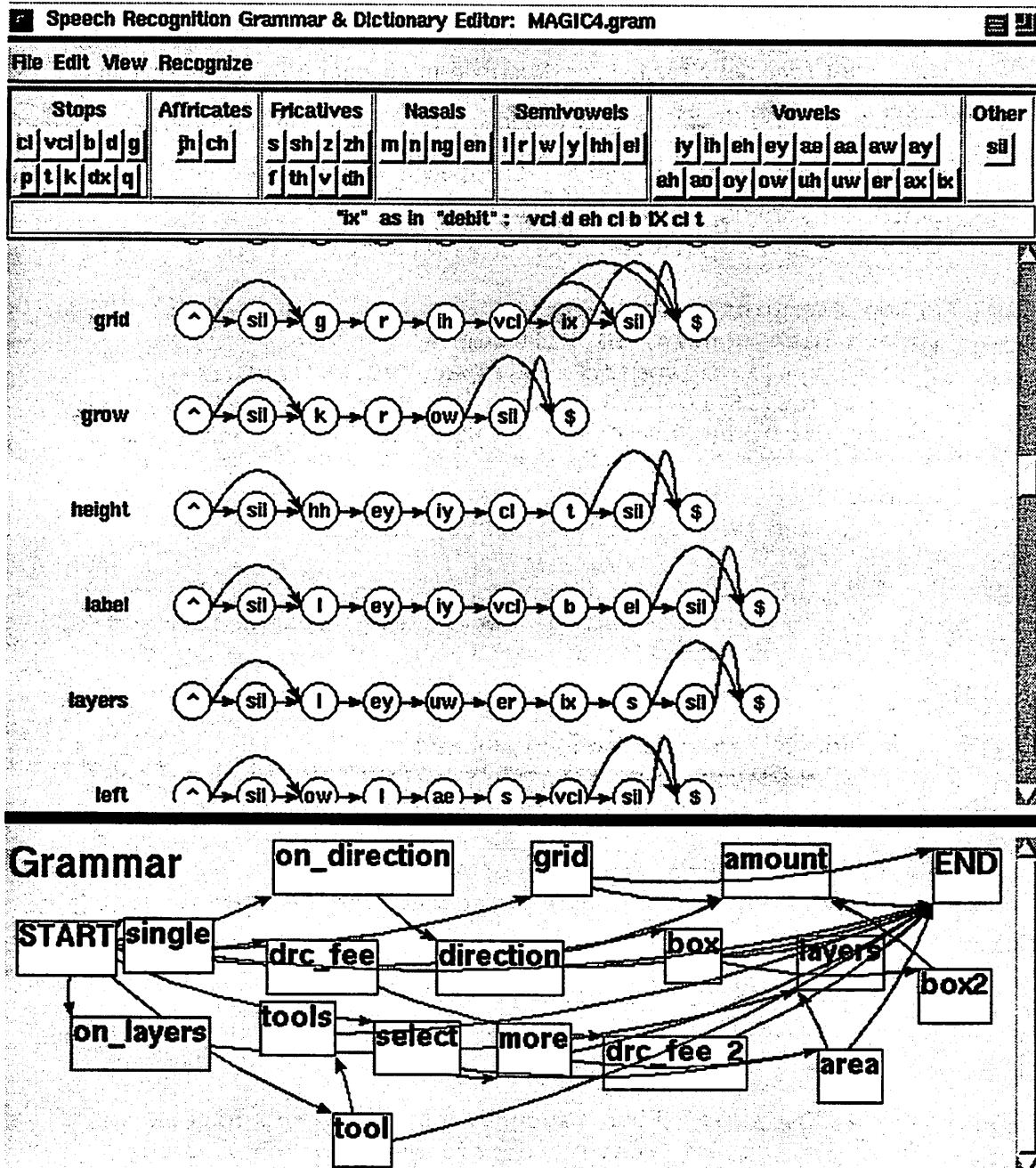


Figure 5-1. GramCracker Application for Creating and Modifying Speech Recognition Vocabularies

to give users access to all necessary controls (e.g. volume control and silence detection levels: see Figure 6-3 on page 121), speaker adaptation, and recognition results. Finally, the programmer can create and edit vocabularies and grammars using the GramCracker application (see Figure 5-1) that allows graphical display of pronunciations and grammars.

5.5.3. Discussion

The circuit schematic recognizer described in Chapter 6 uses this speech recognizer for command entry. It was very easy to add speech recognition capabilities to the application.

The code is illustrated in Figure 5-2. The **speech_dict** object declared in the first few lines

```
source "$SPRCG_LIBRARY/spRcg.tcl"
source "$SPRCG_LIBRARY/spRcg_wigits.tcl"

spRcg_gramDict speech_dict
speech_dict config -mlpFileName \
    "/tools/ui/speechRecog/spRcg/v0.26/libtcl/\
    phone.16.r6.L128.P3.8khz.M0.Norm.hard.weightsq.mlp"
speech_dict readGramDict "$SCHEMATIC/lib/speech.gram

proc add_speech {} {
    global env SCHEMATIC SPRCG_LIBRARY

    if {[info commands speech_record] != ""} {
        .speech.control recordRecog speech_process
        return
    }

    spRcg_recorder speech_record
    if {[catch {set display $env(AUDIOFILE)}] != 0} {
        set display $env(DISPLAY)
    }
    if { [ catch {speech_record open $display} ] != 0 } {
        tkerror "Couldn't connect to AF server at $display"
    }

    eval spRcg_recognizer speech_recog \
        [speech_dict giveRecogConstructorArgs]

    toplevel .speech
    spRcg_controller .speech.control -recognizerName \
        speech_recog -recorderName speech_record
    pack .speech.control

    .speech.control recordRecog speech_process
}
```

Figure 5-2. Code Fragment to Illustrate API for Speech Recognizer

is a dictionary object which reads vocabulary and grammar information created in advance using the GramCracker application. The **add_speech** procedure is called when the user wants to start using speech. It declares the **speech_record** object which records speech and

stores it for later processing, and the `speech_recognizer` object which takes recorded speech and recognizes it. The `.speech.control` window (mega-widget) shown in Figure 6-3 on page 121 is also created.

This API is very simple to use. Other mega widgets allow the user to adaptively train the recognizer, to view the top four recognition candidates, and turn the audio recorder on and off.

5.6. A Real Time Large Vocabulary Speaker Independent Speech Recognizer

The recognizer described in Section 5.5 above is suitable for small-vocabulary applications such as command and control of applications. It is not suitable for dictation. In this section, we describe a recognizer that we built to support dictation. It has a large vocabulary (60,000 words), a statistical grammar (perplexity 60), is speaker-independent, and runs in real time on custom hardware.

This recognizer is an excellent basis for building the remote speech recognition server described in Section 5.4.3. Although it could have been used as a server, it was not connected to the InfoPad system because it is now out of date. It serves as an example of how we could build custom hardware to implement sophisticated algorithms in real time and connect to such a hardware server over the network.

The system is implemented as several custom boards in a VME card cage, and has a host CPU board and an Ethernet board. It is therefore connected to the network. The CPU board runs the VxWorks real-time operation system and can run applications written in C. Therefore the server front end software can run on the CPU board and clients can connect via the network. A collection of such servers can reside on the network to provide real time large vocabulary speech recognition service.

In this case, network bandwidth requirements would not be high. Since the system takes digitized audio quantized to 16 bits at 16 kHz, the total data rate into the recognizer is 256 kbytes/s per user. The data rate coming out of the recognizer is negligible. Network latency and jitter do not affect performance. However lossless network transmission is required.

The current InfoPad terminal provides 8-bit μ -law audio at 8 kHz which is not sufficient for this recognizer to work well. However, future versions of the terminal will have 16-bit 16-kHz audio.

In this section, we explain the recognition algorithm used, including the hidden Markov model used in this speech recognizer. It is significantly different from the model used in the handwriting recognizer described in Chapter 4: this recognizer uses a grammar and a dictionary. Then we describe the system architecture, concentrating on the Active Word processing. We describe the hardware implementation of the system and the hardware simulation environment that was created to simulate the Viterbi Board which implements most of the recognition algorithm.

5.6.1. Differences from the Small, Flexible Recognizer

This speech recognizer is targeted towards general sentence dictation applications where the vocabulary and grammar are the same for all applications. The vocabulary is large and it is not easy to change the vocabulary or grammar on the fly. In contrast, the recognizer described in Section 5.5 allows easy configuration of the vocabulary and grammar. It supports only a small number of words at the same time. The algorithms are also different.

The VLSI recognizer uses triphone modeling and a statistical grammar, whereas the small, flexible recognizer uses phoneme modeling and a rule-based grammar which can be changed on the fly during normal operation.

The VLSI recognizer uses HMM training to calculate output probability distributions whereas the small, flexible recognizer uses a multi-layer perceptron for this purpose. The small recognizer also uses special processing of the audio input to obtain some degree of tolerance of noise and channel distortion.

5.6.2. The Recognition Algorithm

The recognition algorithm has three steps: feature extraction, Viterbi search and backtrace, as explained in the following sections.

5.6.2.1. Feature Extraction

We implemented a non-parametric feature extraction algorithm similar to the algorithm used in the DECIPHER system [Murv89]. A block diagram of the algorithm is shown in

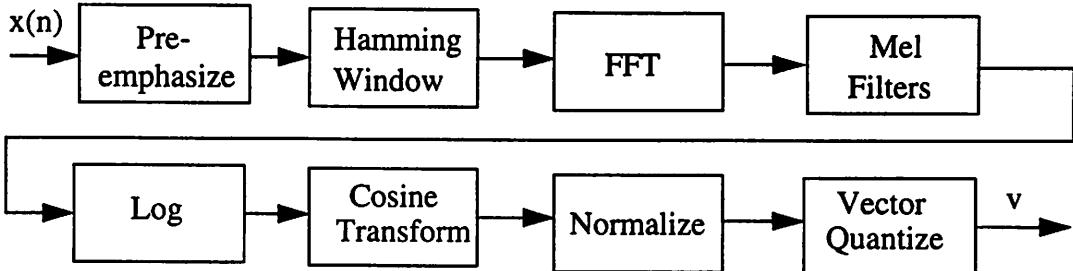


Figure 5-3. Block Diagram of the Feature Extraction Algorithm

Figure 5-3. Incoming speech is sampled at 16 kHz and linearly quantized to 16 bits. It is then pre-emphasized using the following equation:

$$\text{Equation 5-1.} \quad y_n = x_n - (0.95 \cdot x_{n-1})$$

The pre-emphasized speech is then blocked into frames of 512 samples which are spaced 160 samples apart: the frames overlap by 352 samples. Each frame is smoothed with a Hamming window:

$$\text{Equation 5-2.} \quad h_i = m_i \cdot y_{kM-1}$$

where $m_i = 0.54 - 0.46 \cos\left(\frac{2\pi i}{N-1}\right)$ and $i = 0 \dots N-1$.

The resulting frame is then used to compute a 256-point Fast Fourier Transform (FFT). The power in the FFT is integrated using 25 mel-spaced band-pass filters and the energy from each of the mel filters is converted to its logarithm. These log energy values are passed through a cosine transform to get the 13-dimensional mel-cepstrum coefficient vector. The mel-cepstrum coefficient vector is then normalized with respect to its mean and variance, and the 12 higher order elements of the 13 elements of the resulting normalized vector are vector-quantized with a codebook of size 256×12 , which produces an 8-bit feature o_i^1 .

Another feature, o_i^2 , is the vector quantized difference of cepstral vectors. o_i^3 is the scalar quantized energy of the remaining cepstral vector element, and o_i^4 is the scalar quantized differenced energy of this cepstral vector element. Each of these features are represented using 8 bits and are used to match the incoming speech to the states in the Markov model. See [Stol92] for a more detailed explanation.

5.6.2.2. Hidden Markov Modeling

Figure 5-4 shows a graphical representation of a left-to-right hidden Markov model

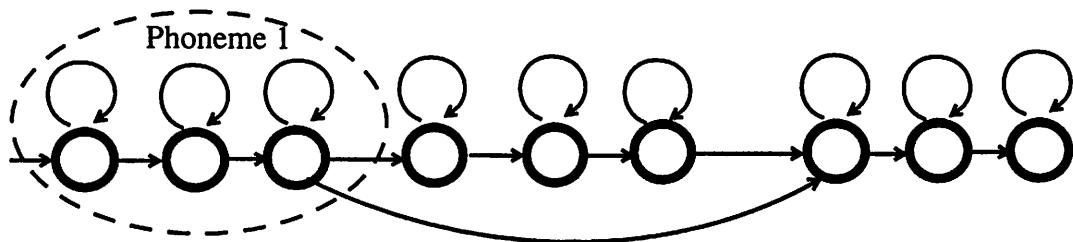


Figure 5-4. Graphical Representation of a Hidden Markov Model

(HMM). In HMM-based speech recognition, speech is assumed to be produced by a hidden Markov process (see Section 4.6.1. on page 54). Each circle represents a Markov state, and each arrow represents an allowed transition between states.

Just as a speech utterance progresses from one sound (such as a phoneme) to another, a Markov model progresses from one state to another. It may stay in the same state for more than one frame, which is illustrated by the self-loops in the Figure. This corresponds to the same sound lasting more than one frame and is therefore a way of modeling the duration of the sound. This model represents speech as having sudden transitions from one state to another, whereas in actuality the transition from one phoneme to another can be gradual.

The model is left-to-right since a spoken word proceeds in a predictable manner from one sound to the next. Some words have optional phonemes, and the transition that skips the omitted phoneme is shown as the arrow from the end of the first phoneme to the beginning of the last phoneme. In our implementation, each state can have at most 3 predecessors, which are defined as preceding states which are allowed to transition to the current state.

Each state transition has an associated transition probability. The feature vector corresponding to each state depends probabilistically on that state. The model is “hidden” because we can observe only the probabilistically dependent features rather than the states themselves, and we have to use the sequence of these features to infer the most probable state sequence, which is an approximation to inferring the most probable word sequence.

In this algorithm, the smallest speech unit modeled is the phoneme. We model each phoneme with a 3-state Markov model to allow for co-articulation effects with the previous and the next phoneme. A word is a concatenation of phonemes. A sentence is a concatenation of words. These concatenations may be done probabilistically and are, in fact, modeled as HMMs. The HMM representation is therefore hierarchical.

For a more general description of hidden Markov modeling applied to speech recognition, refer to Section 4.6.1. on page 54, or to [Juan84].

5.6.2.3. Viterbi Algorithm

The Viterbi algorithm [Juan84] is an approximation to the forward algorithm (see Equation 4-7), which is the optimum maximum likelihood state sequence detector for a HMM. The algorithm calculates the probability of a match with the current utterance for all legal sequences in the model and determines the most probable state sequence. Pruning may be used to discard highly unlikely sequences early in the calculation and thereby reduce computational requirements.

Figure 5-5 shows the trellis used for Viterbi decoding of the HMM. Each circle represents

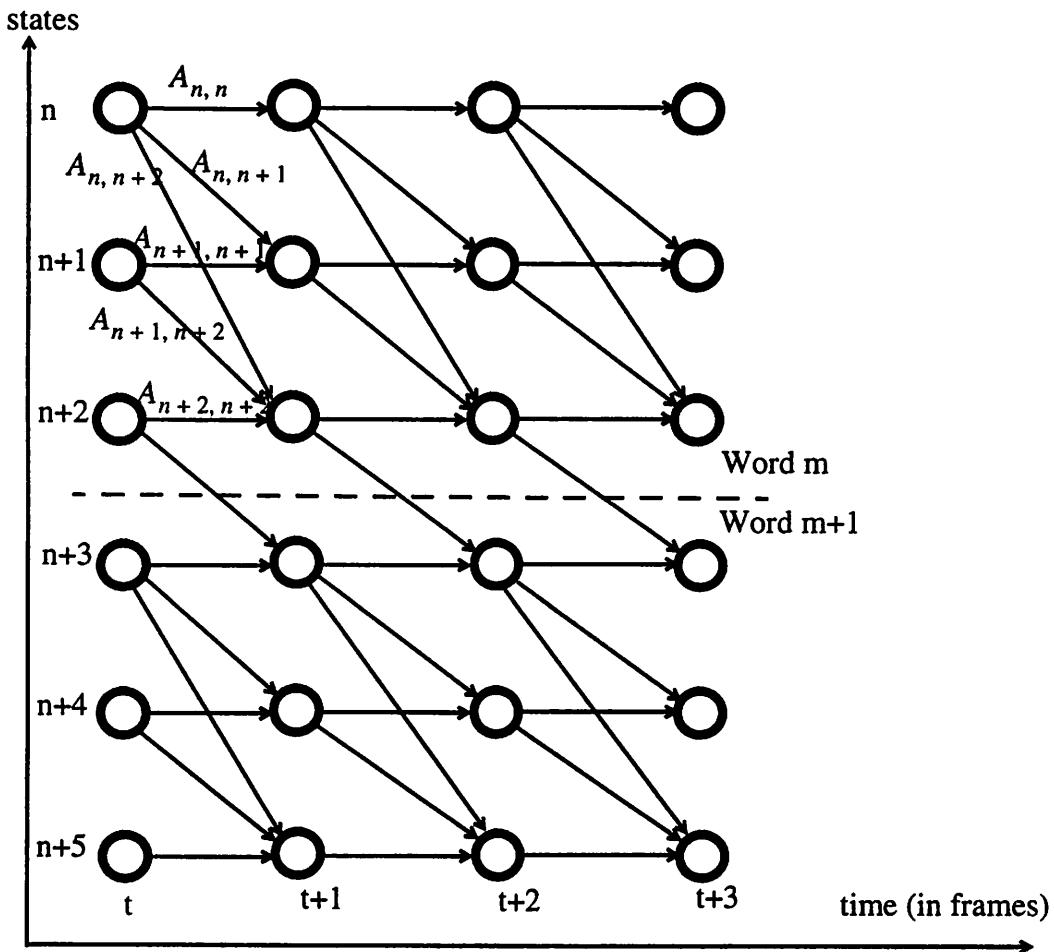


Figure 5-5. Trellis for Decoding a Hidden Markov Model

a Markov state. The horizontal arrows represent the self-loops illustrated in Figure 5-4. Other arrows represent legal state transitions. The parameters $\{A_{ij}\}$ are the state transition probabilities. The horizontal dotted line represents the boundary between two phonemes. Note that in this case, transitions between phonemes are allowed from the last state of the predecessor to the first state of the successor only, these states being the grammar nodes described in Section 5.6.3.2. In a more general case, any state can transition to any other state.

At each frame, the probability of being in each state is calculated using transition probabilities from each predecessor for that state and the feature vector calculated from the current frame of speech. This state probability is calculated for all active states; the probability that

any legal state sequence $S_N = \{s_1 \dots s_N\}$ matches any feature vector sequence $O_N = \{o_1 \dots o_N\}$ may be determined using the following equation:

$$\text{Equation 5-3.} \quad P(S_N, O_N) = \pi(s_1)P(o_1|s_1) \prod_{i=2}^N [A(s_{i-1}, s_i) \cdot P(o_i|s_i)]$$

where $A(s_i, s_j)$ is the transition probability from state s_i to state s_j and $\pi(s_i)$ is the a priori probability that the sentence starts with state s_i . Note that:

$$\text{Equation 5-4.} \quad P(o_i|s) = P(o_i^1|s) \cdot P(o_i^2|s) \cdot P(o_i^3|s) \cdot P(o_i^4|s)$$

where each of the terms is defined in Section 5.6.2.1 and assumed to be independent of the others. However this calculation is very expensive. The number of possible state sequences is $(mN)^T$ where m is the geometric average of the number of transitions into each state, N is the number of Markov states in the system, and T is the number of frames in the sentence. For each second of speech, $T = 100$. For a 60,000 word vocabulary with an average of 15 Markov states per word, $N = 900,000$. If we assume that $m = 3$, then we have to evaluate Equation 5-3 a total of 1.37×10^{643} times, which is clearly a ridiculously large amount of computation.

Since we are interested only in the most probable state sequence leading up to each state, we can omit the calculation of the less probable paths into each state. The Viterbi algorithm is a dynamic programming scheme that uses this approach:

$$\text{Equation 5-5.} \quad P(O_i, s) = \text{MAX}_{p \in \{pred\}} [P(O_{i-1}, p) \cdot A(p, s)] \cdot P(o_i|s)$$

where the initial condition is

$$\text{Equation 5-6.} \quad P(O_1, s) = \pi(s) \cdot P(o_1|s)$$

and the set $\{pred\}$ is the set of predecessors of the state s . $\pi(s)$ is the a priori probability that the state s is the beginning of a sentence. This equation is calculated mNT times, which corresponds to 270 million times per second. This number can be reduced by pruning. Pruning eliminates improbable paths from the search space, processing only the highly

probable paths. A pruning threshold is used to determine which candidate paths to keep, as explained in Section 5.6.3.1 below. Pruning can increase the error rate by eliminating the correct path early. However experiments can be performed to determine a good pruning threshold for any given recognizer.

5.6.2.4. Backtrace

At the end of the sentence, after Equation 5-5 has been calculated for the last frame, we need a means of tracing the path from the state at the end of the most probable sequence back through the trellis to the first frame so that we can report the entire sequence. This is done by maintaining a backtrace list where, for every state and for every frame, a tag pointing to the most probable preceding state is stored. This tag is the address in the backtrace list of the preceding state. Using these tags, the most likely state sequence can be determined at the end of the sentence.

5.6.3. System Architecture

The algorithm described in Section 5.6.2 above was implemented in custom hardware. This section describes the architecture and implementation of the system.

5.6.3.1. Changes to Improve Performance

Some changes were made to the algorithm to improve performance. In Equation 5-5 above, the multiply operations can be replaced by additions if logarithmic representations of each term in the equation are used.

This algorithm is ideally suited to floating point representations, but such a representation would result in complex floating point datapaths and to larger memories. To avoid this increased complexity, we implemented a fixed-point representation with frequent normalization to retain accuracy and avoid underflow. Normalization is done using the following equation:

Equation 5-7.

$$P_n(O_i, s) = \frac{P(O_i, s)}{\text{MAX}_{t \in S}[P_n(O_{i-1}, t)]}$$

where $\text{MAX}_t[P_n(O_{i-1}, t)]$ is the largest normalized state probability from the previous frame, and S is the set of all states. This division operation is expensive to implement in hardware. However, the logarithmic representation reduces it to a subtraction.

A pruning scheme helps reduce the computational requirements of the system by discarding word models whose states have low state probabilities. A pruning threshold Θ is computed before every frame as follows:

$$\text{Equation 5-8.} \quad \Theta_{\text{new}} = \text{MAX}[\Theta_{\text{old}}, \text{MAX}_{s \in S} P(O_i, s) \cdot \Theta_{\text{offset}}]$$

where Θ_{offset} is a system parameter. Pruning is used to reduce the number of active states by a factor of 5.

5.6.3.2. Hierarchy

The Viterbi recursion (Equation 5-5) is implemented on 2 levels of hierarchy, phone processing and grammar processing. In phone processing, the probabilities of states within phoneme models are updated using only transitions from within the same phoneme. In grammar processing, the transition probabilities between phonemes are used to compute the probabilities of states within successor phonemes.

The advantage of this hierarchical scheme is that both levels of processing can be performed in parallel. However there is a large amount of data that is passed between the two levels since it is possible for transitions to successor phonemes to begin from more than one state in each predecessor phoneme, and transitions from predecessor phonemes can end in more than one successor phoneme.

To reduce the data rate between the two levels of hierarchy, we define two artificial nodes, the *source grammar node* and the *destination grammar node* (see Figure 5-6). The source

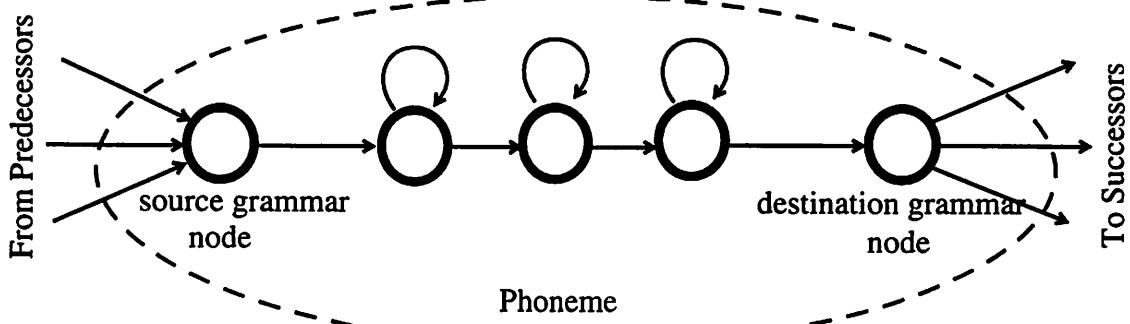


Figure 5-6. Source and Destination Grammar Nodes

grammar node probability is the probability that the state sequence terminates at the beginning of the phone instance, corresponding to the MAX operation in the Viterbi recursion (see Equation 5-3) with each predecessor probability taken as the destination grammar node probability of the predecessor state.

For each phoneme, the destination grammar node probability at frame i is computed as follows:

$$\text{Equation 5-9.} \quad P(D)^i = \text{MAX}_{p \in \{\text{pred}\}} [P(O_i, p) \cdot A(p, D)]$$

where $\{\text{pred}\}$ is the set of states within the phoneme which have transitions to the destination grammar node. To prevent overflow, the destination grammar node probabilities are normalized after every frame. The destination grammar node probabilities must be computed before the source grammar node probabilities.

This scheme also helps reduce the memory requirements of the backtrace algorithm. Instead of storing a backtrace tag for every state in every phoneme for every frame, we can store a tag for each destination grammar node only, with the tag pointing to the predecessor's destination grammar node. Whenever the probability of the destination grammar node is high enough to be potentially part of the most likely path, it is stored in the backtrace list (together with a tag pointing to its predecessor phoneme's destination grammar

node) and its tag is passed on to the source grammar nodes of its successors. The source grammar nodes pass the tag along to their destination grammar nodes as part of the normal Viterbi processing.

5.6.3.3. System Hardware Partitioning

The hardware partitioning of the system is illustrated in Figure 5-7. All the custom boards

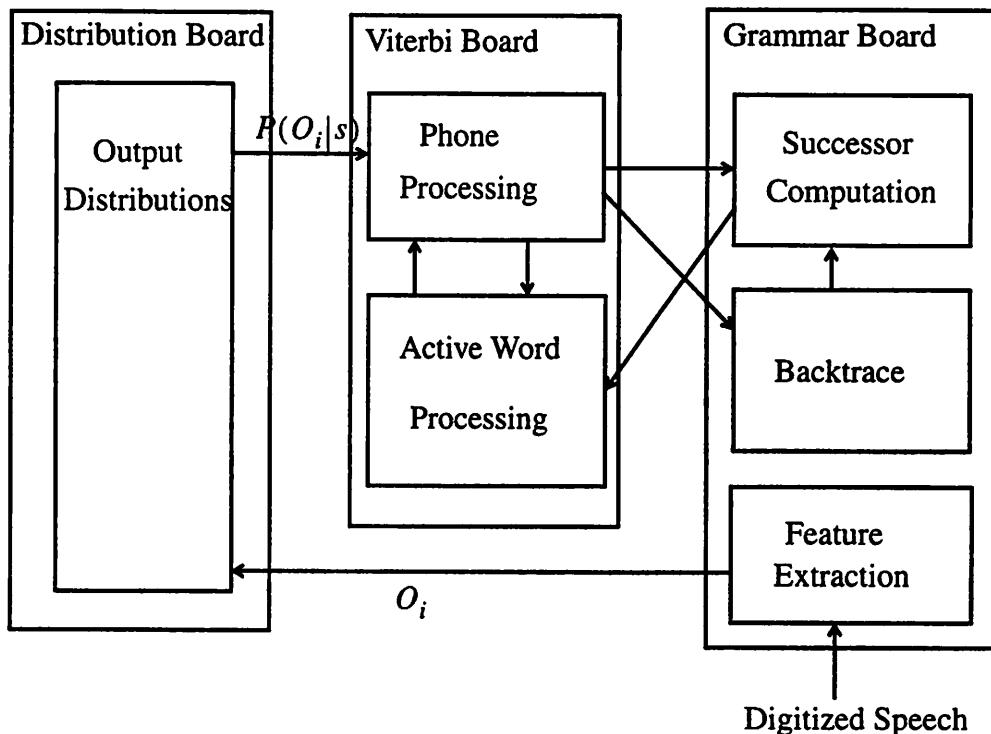


Figure 5-7. Hardware Partitioning of the Speech Recognition System

sit in a VME card-cage together with a CPU board and an Ethernet board.

Feature extraction is done on the Grammar Board, and the feature vector is sent to the Distribution Board via the VME backplane. The set of probability distributions corresponding to that feature vector is sent from the Distribution Board to a cache on the Viterbi Board via a ribbon cable. On the Viterbi Board, the Phone Process updates the state probabilities of all the active states. If any of the state probabilities of an active phone has a probability higher than the threshold, the Phone Process sends a message to the Active Word Process to put this instance in the Active Word List for the next frame. If the destination grammar node probability of a phoneme is high, the Viterbi Process tells the Successor Computation

Process on the Grammar Board to generate a backtrace list and send successors to the Active Word Process. The Active Word Process maintains the Active Word List. Communication between the Viterbi Board and the Grammar Board is done via ribbon cables. At the end of a sentence, the Backtrace Process on the Grammar Board generates a list of recognized words.

A Heuricon 68020-based general purpose microprocessor board and an Ethernet interface board are used to control the system and to interface with the network. These boards allow start-up and parameter loading of the custom boards and synchronizes the boards after every frame. They also allow access from a UNIX workstation on the network to the memories on the custom boards for operational and debugging purposes.

5.6.4. Phone Process

Phone processing involves calculating the state probabilities for all the active states including the destination grammar node probabilities but excluding the source grammar node probabilities. The source grammar node probabilities are calculated by the Successor Computation Process on the Grammar Board.

The Phone Process is implemented in 3 custom VLSI chips. For more details on its implementation, see [Stol92].

5.6.5. Active Word Process

The Active Word Process generates and maintains a list of active phone instances for the next frame. It takes input from the Phone Process and the Successor Computation Process, as illustrated in Figure 5-8. We use the terms “word”, “phone” and “phone instance” interchangeably in the rest of this Section since each item is defined by the existence of a source and a destination grammar node. As long as the item, be it a word or a phoneme, has these grammar nodes the Active Word Process does not distinguish between them. For example, if the recognizer were phoneme based then each item would have three states including the grammar nodes whereas if the recognizer were word based then each item would have an average of 17 states. This speech recognition system is triphone based, which means that each subword unit models coarticulation effects with adjacent subword units, resulting in three HMM states per triphone, excluding the grammar nodes.

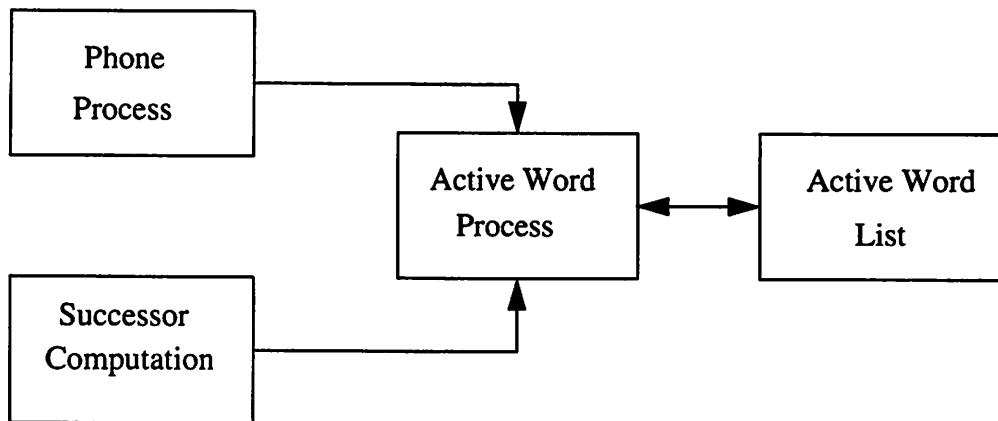


Figure 5-8. Connections to the Active Word Process

5.6.5.1. Description

The Phone Process supplies a list of words (actually phonemes) that are currently active and should remain active during the next frame. These are words which have at least one state with high probability. The Successor Computation Process supplies a list of new words that are successors to the words which are currently being processed by the Phone Process and which are ending. This latter list may have redundant words; in this case, the Active Word Process performs a MAX operation on the source grammar node probabilities and stores only the most probable one in the Active Word List.

The fields in the Active Word List are listed in Table 5-2.

Name	Length (bits)	Comment
P(S)	16	source grammar node probability
Tag(S)	20	source grammar node tag
WordArc	20	phone instance identification
StProbAdd	18	address in the state probability memory
UniqueAdd	16	address of the unique phone
TopoAdd	4	address in the topology memory
NewFlag	1	flag to indicate that the phone instance is new
EndFlag	1	flag to indicate the end of the Active Word List

Table 5-2. Contents of the Active Word List

$P(S)$ is the probability of the source grammar node of this word. It is used only when the word was supplied by the Successor Computation Process. If the word was supplied by the Phone Process, this probability is set to 0.

$Tag(S)$ is the backtrace tag generated by the Successor Computation Process. If the word was not generated by the Successor Computation Process, $Tag(S)$ is meaningless.

$WordArc$ is the ID of the word (or phoneme) instance, and is unique for each instance of that phoneme.

$StProbAdd$ is the address of the source grammar node of this word instance in the State Probability Memory that is part of the Phone Process, and is valid only if this word was supplied by the Phone Process.

$UniqueAdd$ is the address of the prototype (or unique) phoneme in a phoneme based recognizer. The idea behind this is that there are a number of unique phonemes which have unique topologies and topology probability distributions, and instances of each unique phoneme would share the topology and probability distributions. Each instance is part of a different word and be processed separately by the Phone Process with a different ID, which is stored in $WordArc$ above.

$TopoAdd$ is the address in the Topology Memory (which is part of the Phone Process) of the unique phone corresponding to the current phone.

$NewFlag$ is set to 0 if the word was supplied by the Phone Process, even if the Successor Computation Process also supplied the same word. $NewFlag$ is 1 only if the word was not supplied by the Phone Process, and this flag tells the Phone Process to ignore the contents of the $StProbAdd$ field.

$EndFlag$ is set to 1 at the end of the Active Word List to tell the Phone Process to stop processing the current frame. It is 0 for all valid entries.

5.6.5.2. Implementation

The Active Word Process is implemented in 3 custom chips and uses 2 memories, as shown in Figure 5-9. It is necessary to use 3 chips because of the large pin count of the

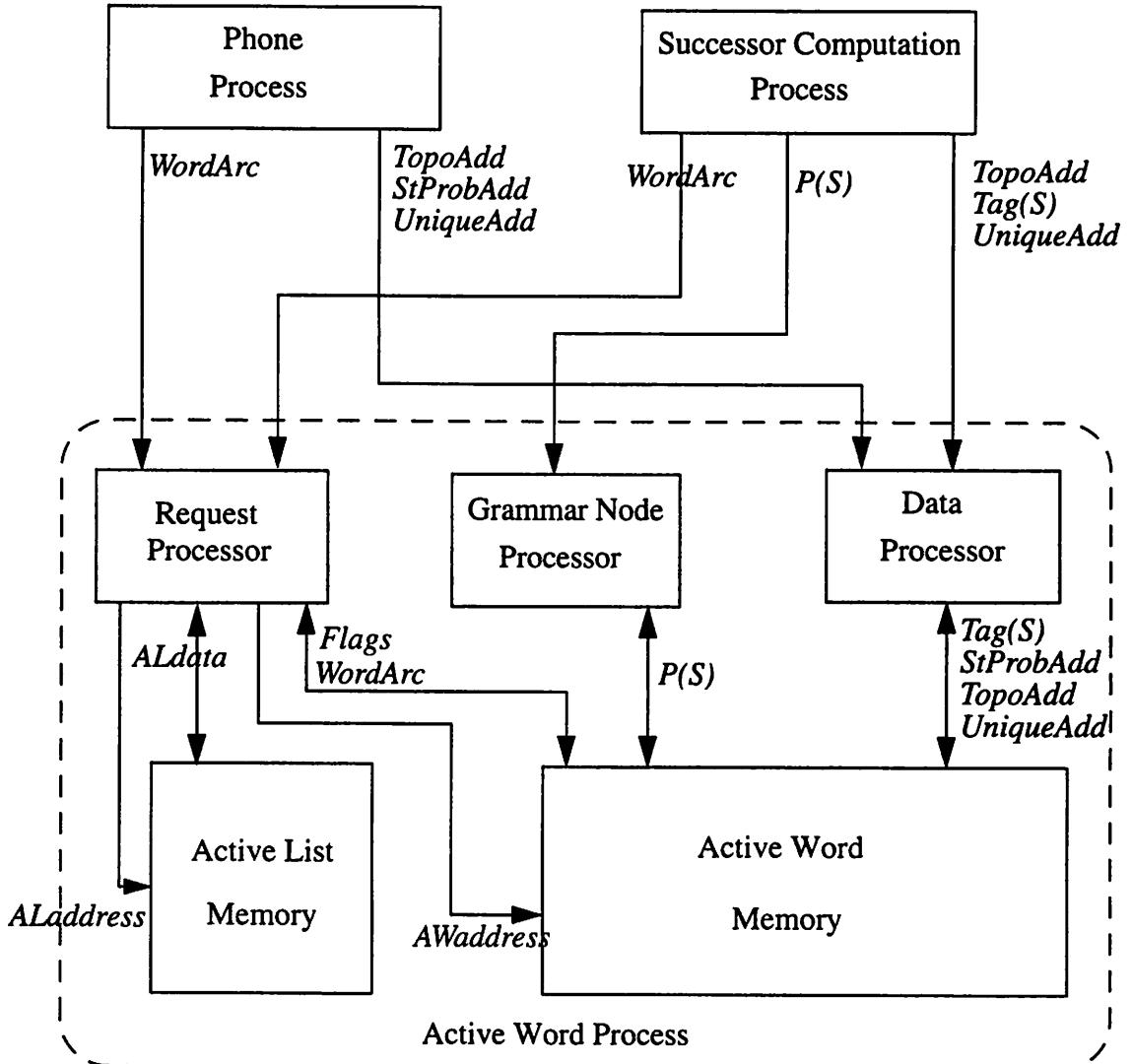


Figure 5-9. Implementation of the Active Word Process

Active Word Process. The main controller for the Active Word Process is on the Request Processor, and the data paths for the fields in the Active Word Memory are bit-sliced across the 3 chips. All operations involving the Active List Memory are also done on the Request Processor. The chips are described in greater detail in Section 5.6.5.3 through Section 5.6.5.5.

It is necessary to use 2 memories for the Active Word List to reduce the actual memory requirements for storing and maintaining the List. The Active List Memory (1 MWords by 16 bits) is addressed using the *WordArc* identifier and contains an address that is used for access into the Active Word Memory (64 kWords by 96 bits). This indirect addressing scheme uses 2.75 Mbytes of memory whereas a direct addressing scheme would use a single 1 MWords by 96 bits Active Word Memory, which would require 12 MBytes of memory.

All entries in the Active List Memory must be zeroed out before processing begins. Whenever the Active Word Process receives a request from the Phone Process or the Successor Computation Process, it reads the Active List Memory to see if the word is already in the Active Word Memory. If so, the old data is retrieved and merged appropriately with the new data, as explained in Section 5.6.5.1, and then re-written into the Active Word Memory. If the word is not already in the Active Word Memory, it is written into the Active Word Memory at the next available address and this address is stored in the Active List Memory at the appropriate location.

5.6.5.3. Request Processor

A chip photograph of the Request Processor is shown in Figure 5-10. It is packaged in a

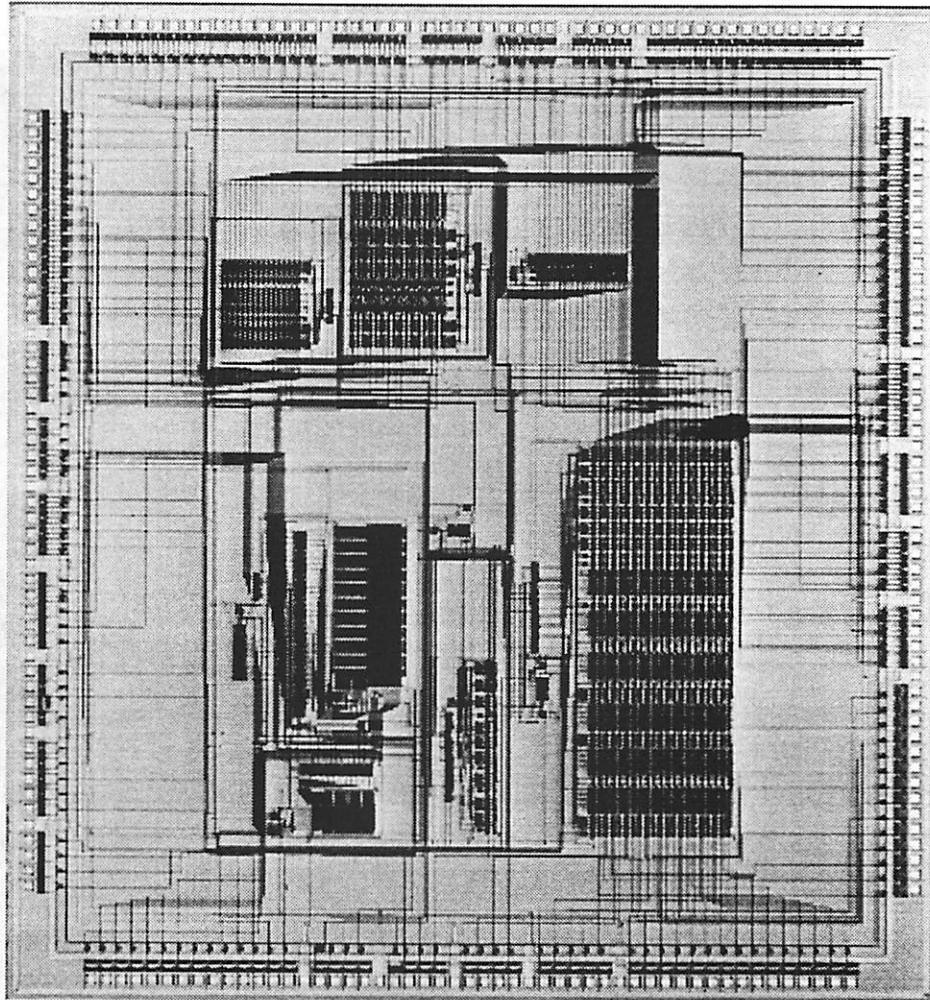


Figure 5-10. Chip Photograph of the Request Processor

208-pin Pin Grid Array (PGA). The controller for the Active Word Process is on this chip, as is the datapath for the *WordArc* processing and for *NewFlag* and *EndFlag*. All circuitry for managing the Active List Memory is on the Request Processor.

5.6.5.4. Probability Processor

A chip photograph of the Probability Processor is shown in Figure 5-11. It is packaged in

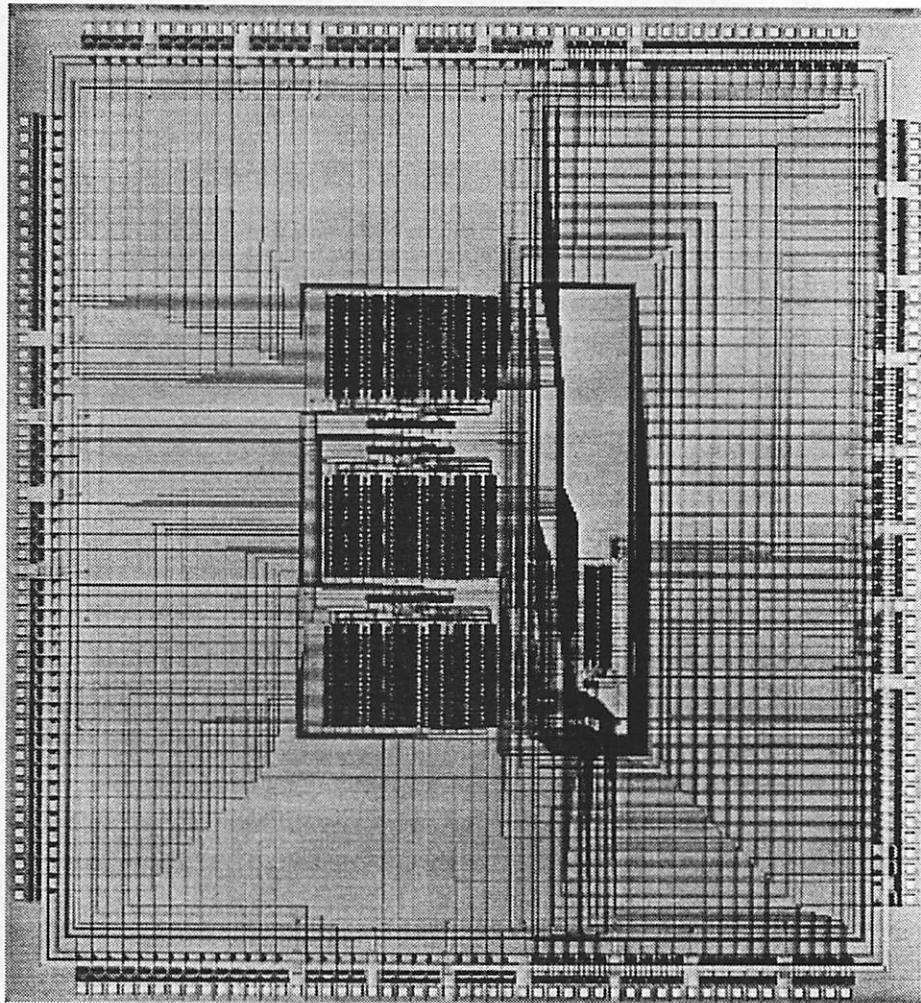


Figure 5-11. Chip Photograph of the Probability Processor

a 208-pin PGA, and runs as a slave to the Request Processor. The Probability Processor contains the datapaths for *StProbAdd*, *Tag(S)*, *TopoAdd*, and *UniqueAdd*.

5.6.5.5. Grammar Node Processor

A chip photograph of the Grammar Node Processor is shown in Figure 5-12. It is packaged

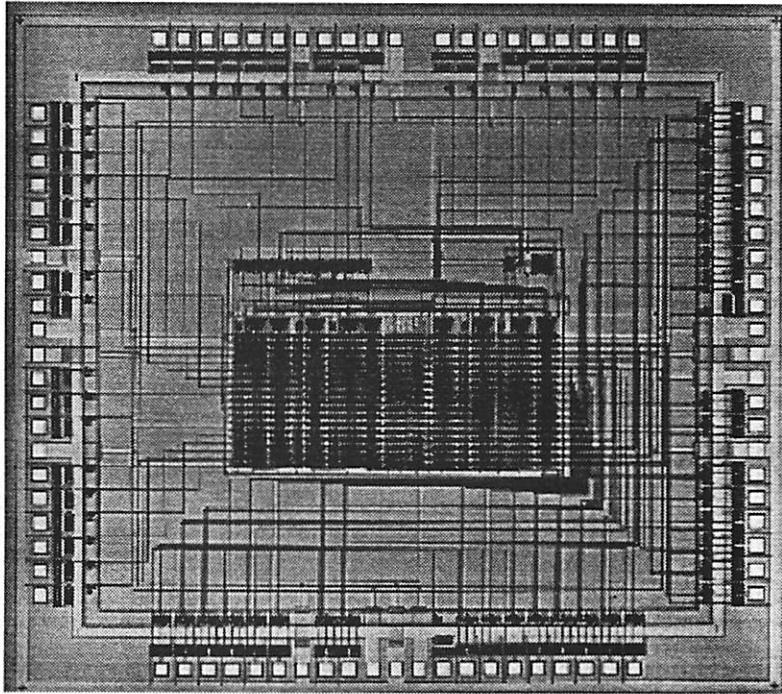


Figure 5-12. Chip Photograph of the Grammar Node Processor

in a 84-pin PGA and runs as a slave to the Request Processor. The Grammar Node Processor contains the datapath for $P(S)$.

5.6.6. Viterbi Board Design

The architecture of the Viterbi Board is very complex. There are two State Probability Memories and two Active Word Memories which are accessed during every frame. There are also two Output Probabilities, one of which is being read by the Phone Process while the other is pre-loaded from the Distribution Board. These three pairs of memories switch function every frame. This switching means that there must be multiplexers on the memory buses. Integrating these buses on the chips means an impossibly large number of pins per chip, and putting the multiplexers on the board means a large chip count and an even more complex board.

The Viterbi Board was therefore designed with a switching architecture. In Figure 5-13, only one half of the board is shown. The other half is a mirror image: each of the memories

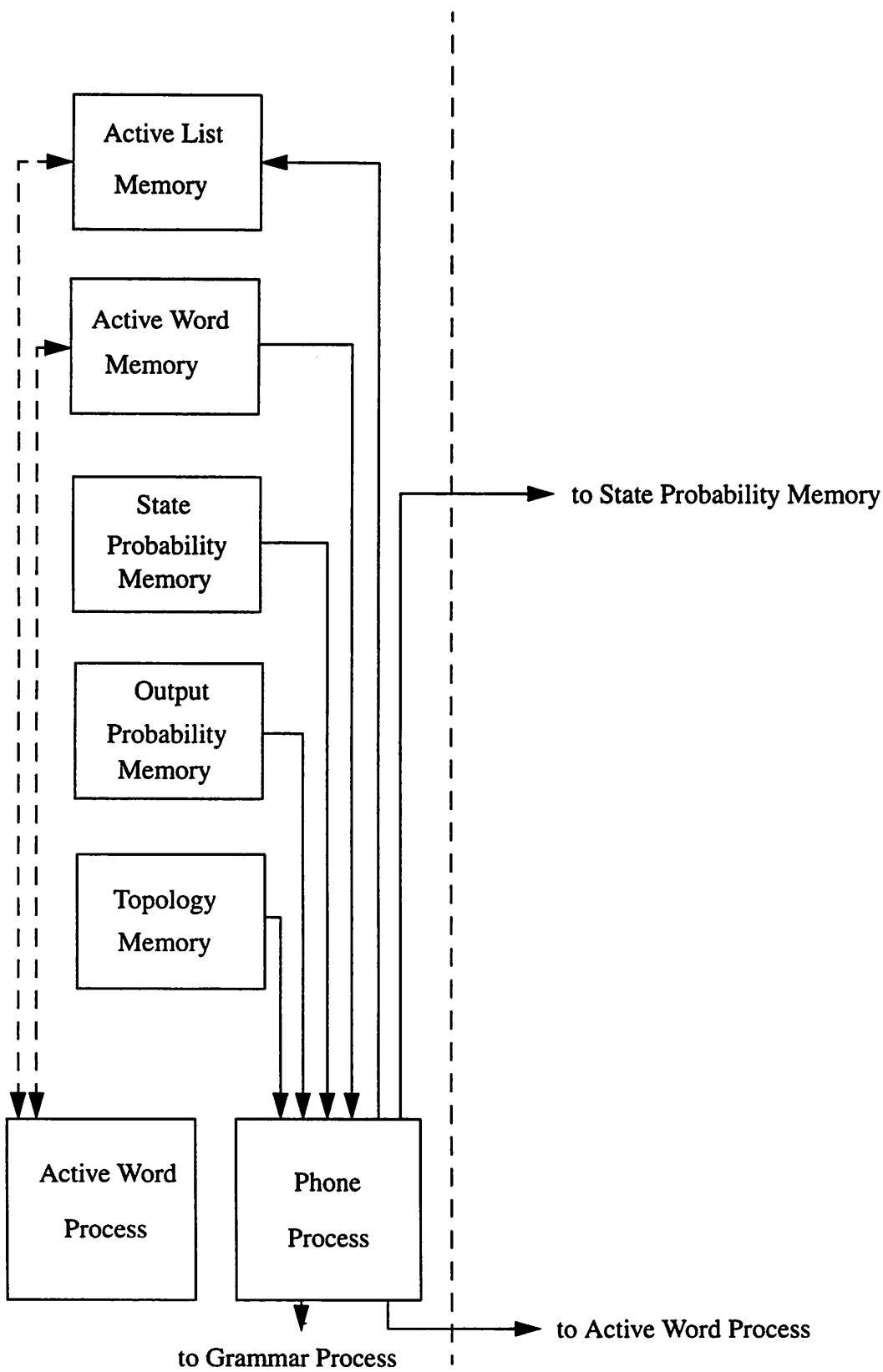


Figure 5-13. Switching Architecture of the Viterbi Board

and custom chips is instantiated twice. Memories may be accessed by the VME interface only through the custom chips. During a frame the Phone Process reads from the Active Word Memory, the State Probability Memory, the Output Probability Memory, and the Topology Memory on its side of the board while the Active Word Process updates the Active List Memory and the Active Word Memory on the other side of the board. The Phone Process also clears the Active List Memory on its side of the board. On the next frame, the process is mirrored on the other side of the board.

The Viterbi Board design has several large memories and many other components. They are listed in Table 5-3 and Table 5-4. The board occupies a triple-height full-depth VME

Memory	Words	Bits/Word
Active List Memory	1 M	16
Active Word Memory	64 k	96
State Probability Memory	256 k	36
Output Probability Memory	64 k	12
Topology Memory	64 k	16

Table 5-3. Memory Sizes on the Viterbi Board

printed circuit board. The design was created using the Lager system [ERL88] and the board was laid out and routed using Racal-Redac's Visula software [RR90].

5.6.7. System Level Simulation

Given that the board is large and complex, it was desirable to simulate its functionality to the extent possible before actual fabrication and test. Simulation is not usually used to verify printed circuit boards. Instead, expert board designers rely on their design skills to get the design mostly right the first time and then use their debugging skills to get it working. But after the experience of VLSI chip design I felt that board level simulations would help the design and debugging process and reduce the time it would take to get it working. I had to create the board simulation environment from scratch since there was no existing capability for such an effort.

There are two types of simulations that may be done on a board design. A timing simulation can uncover all connectivity, functionality and timing problems, but requires timing

Part Type	Quantity
22V10 PLD	14
16L8-7PC PLD	5
74AS244 buffer	77
74AS374 flip-flops	6
VME2000 VME Interface	1
VME3000 VME Interface	1
74AS646 transceivers	3
74AS74 flip-flops	2
EP610 PLD	1
74AS04 inverters	1
74AS02 or-gates	1
20 MHz oscillator	1
M1641 SIP memory	14
M1831 SIP memory	6
M1621 SIP memory	4

Table 5-4. Components on the Viterbi Board

models for all the components on the board, including the PLDs and the custom chips. I could not find a simulation engine that handles a heterogeneous design like this one. Creating timing models for all the components would have taken too long to be worthwhile.

A second option is a behavioral simulation. In this case, only connectivity and functional problems would be found. This is sufficient since the board is clocked and the interfaces are very well specified. A separate program was created to find high fanout nodes so that buffers could be added.

A public domain event-driven simulator called THOR [Stan86] was used to simulate the board. In THOR each chip is modeled with a behavioral description in a language similar to C. These C-like routines are connected at a higher level using a LISP-like language called CSL. Hierarchical designs are supported via hierarchy of CSL files.

For this simulation, THOR models were written for all the off-the-shelf components on the board and for all the individual components within the custom chips and the PLDs. The simulator was then compiled and run with manually created test vectors. This simulation

was particularly valuable in finding problems in the synchronization between the Phone Process and the Active Word Process and between the custom chips and the memories.

6 Applications

Several applications were built for the InfoPad system. The applications that most involve use of the pen are the Circuit Schematic Recognizer and the Electronic Notebook. The former also uses handwriting and speech recognition. Since the focus of this thesis is the use of pen and speech in the user interface we examine these two applications in this chapter. Some other InfoPad are described in [Nara96].

6.1. Circuit Schematic Recognizer

The Circuit Schematic Recognizer is the only InfoPad application that uses the entire recognition infrastructure and is therefore the showcase application for the InfoPad system. It allows us to explore the use of the pen as a design tool and the synergy between pen and audio input in an application. The pen is used as a drawing tool and for accessing menu driven commands, the handwriting recognizer for entering parameters and labels, and the small, fast speech recognizer described in Section 5.5 for commands. The application recognizes drawn circuit elements and automatically decides whether drawn items are elements or wires.

It is implemented entirely in Tcl [Oust94] and uses the Tk toolkit of X widgets. An object-oriented extension to Tcl called itcl [itc96] was used for ease of programming.

6.1.1. Desired Functionality

This application is designed to be a front end to the SPICE [Meta91] circuit simulation program. Users draw simple electrical circuits containing circuit elements and connecting wires on a drawing canvas in a freehand style using a pen. The application captures this freehand drawing and displays electronic ink on the drawing canvas. It recognizes drawn circuit elements and generates a SPICE deck representing the drawn circuit. Wherever it

makes sense, speech recognition is used to improve the speed or convenience of design entry.

At a higher level, this application is designed to allow the user to enter a circuit schematic design faster and more comfortably than he would be able to using traditional means such as manually creating a netlist file or using a keyboard and mouse with a CAD tool. The design entry process is improved by using pen and speech rather than a keyboard and mouse.

6.1.2. Current Systems

There are currently no commercial or research prototypes of pen-based circuit schematic capture tools available. However there are many tools based on a keyboard and mouse interface [Hass95] [McMu92] [Gill95].

The biggest problem with a keyboard/mouse interface is the constant alternation between mouse and keyboard. The mouse is used to place objects and to select commands from menus. The keyboard is used for keystroke shortcut commands and for entry of parameters and labels. This alternation between input devices is inefficient and annoying. The mouse interface is also inefficient because of the need to move the mouse from the drawing canvas to the menu bar for commands or the palette of circuit primitives for creation of new circuit elements. Mouse interfaces also require that the user create each circuit primitive, then rotate it to the desired orientation, then move it to the required location. That means three operations for each primitive, rather than one.

A pen/speech interface overcomes many of these limitations. The pen and the microphone are used by different parts of the human body, so alternating between the two does not require physical movement. The pen may be used to draw items in place and speech may be used to issue commands that are traditionally menu driven. Pen or speech may be used to replace keystroke shortcuts. Items drawn with a pen can be placed at the correct place with the correct orientation in one operation.

A pen/speech system can therefore provide a much better interface for schematic capture and for a large class of computer aided design applications.

6.1.3. Lessons Learned

Implementing a pen-aware application revealed several unique characteristics of using a pen to drive an application. These characteristics stem from the way people write rather than anything inherent in the Pen Server or the InfoPad environment. They are therefore universal and must be accounted for in any pen-based user interface.

The black-and-white (monochrome) screen of the present InfoPad version means that feedback to the user cannot use color, so bitmaps and wires have to be represented in such a way that they are clearly visible and distinguishable on a monochrome screen.

Users have trouble double-tapping a pen (equivalent to double-clicking a mouse). It is difficult to put the pen down on the same spot twice in quick succession. The second tap usually comes down several screen pixels away. This phenomenon is due to the need to lift the entire forearm for each tap: for a mouse, the user keeps the pointing device steady while pressing buttons with his fingers so this problem does not arise. Also, the pen sometimes bounces. Therefore this application does not use pen double-taps.

The difficulty in tapping the pen accurately also relates to using pull-down menus. Users often tap on a menu button, lift the pen until the menu comes up, then tap the pen on the desired item. But they may miss the intended menu item if they are a little sloppy. Therefore, all the menus are designed to be tap-and-drag. Menus must appear on one side of the pen rather than directly below it, otherwise the pen will obscure them.

Many electronic pens come with two buttons, one on the tip and another on the barrel. However the barrel button is difficult to use. Its position is not intuitive and users often press it accidentally. When called upon to press it deliberately, they often have difficulty locating it. This application therefore does not use the barrel button. Effectively, we have a one-button pen.

The pen is a natural pointing device. Users like to get visual or audio feedback from their activities on screen and it would appear that pen position alone is enough to tell the user which window contains the pen. Since it is physically on the screen, the pen can replace the functionality of the mouse cursor. However, users are uncomfortable without a pen

cursor drawn on the screen, or some other way of telling which window or widget is currently in focus. This discomfort is aggravated by the parallax problem caused by the thickness of the glass covering the LCD screen. Therefore, a pen cursor is visible at all times when the application is active. This feature of keeping the cursor visible even when the pen is up is supported in the Pen Server, described in Section 3.3. on page 32.

The pen generates about 100 points per second and a pen tap is slower than a mouse click due to the mechanical movement of the switch at the pen tip. Therefore, the Pen Server generates a line every time the pen is tapped on the screen. The number of points in the line varies depending on the user's tapping style. Therefore the application has to interpret all drawn lines and determine whether they were intended to be pen taps or lines.

The mechanical delay in the tip switch when the pen is picked up or put down means that when the pen is picked up after a line is drawn, an artifact persists in the direction in which the pen was moving at the time. This artifact, or tail, means that drawn objects, especially those consisting of multiple strokes, often have artifacts at the end of each stroke. The recognizer therefore has to detect and filter out these spurious points.

The built-in geometric object recognizer performs recognition of drawn items when the pen is picked up after strokes have been drawn. Multi-stroke elements are drawn with user-dependent times between pen-up on the previous stroke and pen-down on the next stroke. The recognizer therefore has to support user-programmed recognition time-outs.

Users align drawn items on the screen with varying degrees of precision, at least partly because the pen obscures their view of the screen. Therefore it is sometimes not obvious whether the user intends that two strokes be connected, or if a wire drawn close to a circuit element is meant to be connected to that element. The application therefore has to allow user-dependent alignment distances to allow for user sloppiness.

In noisy environments or when a quiet environment is needed, speech recognition cannot currently be used to drive the application. Noisy environments cause the recognizer to make more recognition errors. Therefore, the speech recognition should be enabled and disabled at will by the user. All functionality can be accessed without speech, although

spoken input can improve the usability of the application. However there are some uses, such as issuing commands, where speech is a better input modality than handwriting. The solution is better, noise-tolerant speech recognizers.

User tests showed that when speech recognition worked well for users, they preferred it to using pull-down menus for issuing commands. When the speech recognizer made frequent recognition errors, users abandoned speech completely in favor of using the pen even if the errors were only on the same small number of utterances.

6.1.4. Implementation

A screen shot of the Circuit Schematic Recognizer is shown in Figure 6-1. The application

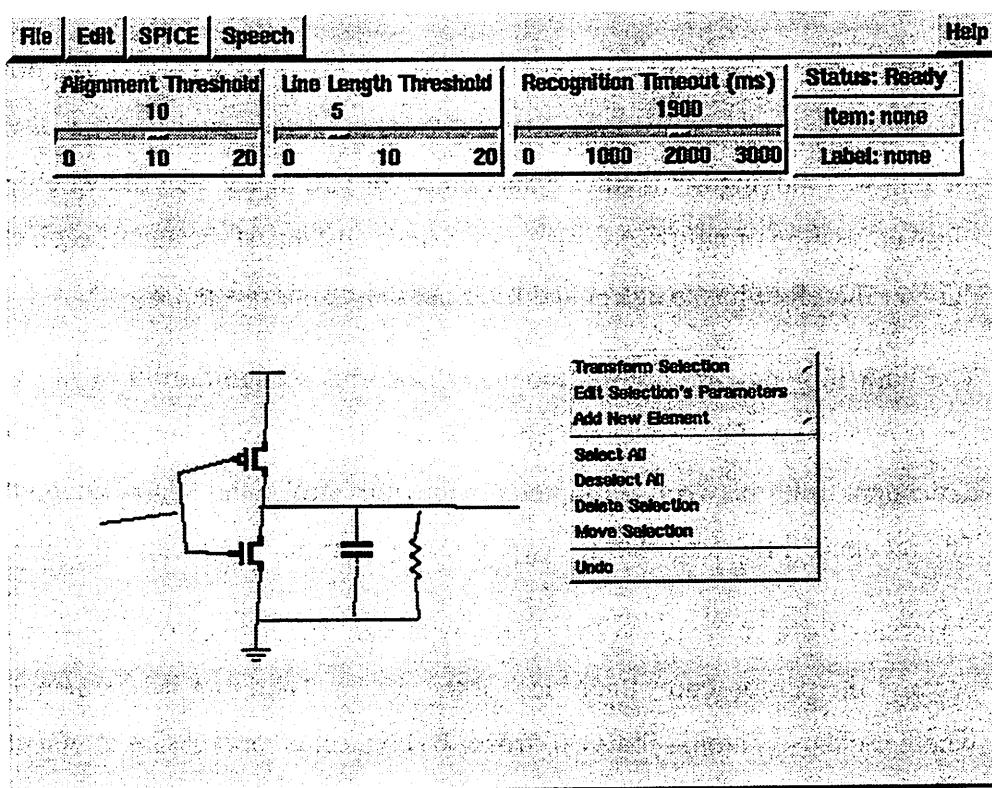


Figure 6-1. Screen Shot of the Circuit Schematic Recognizer with Edit Menu Posted

starts up with a blank canvas, a row of menu buttons, and some control sliders along the top. The **File** menu provides the usual file manipulation commands. The **Edit** menu, also shown posted in the Figure, allows manipulation of objects on the canvas. The **SPICE** menu controls SPICE file generation and the **Speech** menu allows speech recognition to

be enabled and disabled. The **Help** button brings up a manual page describing the application.

In normal operation, the user draws circuit elements and wires. This drawing is electronically inked onto the canvas. The application segments each drawn stroke into lines or circles and then looks for ordered sequences of these primitives to make up circuit elements. The recognition algorithm compares the drawn sequence of primitives to stored templates of primitive sequences representing circuit elements to find a match. This comparison uses a priority ordering of most complex element first. Currently, the application supports NMOS and PMOS transistors, resistors, capacitors, Vdd, and ground circuit elements. It also recognizes rotations and reflections of these elements. If a drawn item is not identified as a circuit element, the application assumes it is a wire or collection of wires.

Recognized circuit elements are displayed with bitmap representations while recognized wires are displayed as straight lines. The lines are not manhattanized because it is not obvious where the user would want to place the additional segment generated by manhattanizing a wire that is almost horizontal or almost vertical. All recognized items, including wires, are aligned with nearby items and are electrically connected to these items.

The “tails” generated by slow switch response on pen-up are dealt with by adding templates incorporating these tails to the library. By observing actual users in action, we were able to generate a list of situations where these tails commonly arise. Slow switch response is also partly responsible for pen taps producing short lines on the screen. The application detects short lines and interprets them as pen taps.

In this application, the recognition algorithm is very simple and completes in a fraction of a second. However, to allow for more sophisticated recognition in future, there is a status indicator on the top-right corner of the main window which says “Busy” during recognition. It is also an indication to the user that recognition has begun so the screen display is being updated.

Items on the screen, including circuit elements and wires, may be selected by tapping the pen on the item. Selected circuit elements appear in inverse video while selected wires

appear as dotted lines. This representation was chosen because the InfoPad has a monochrome screen which does not allow visual feedback in color. Many items may be selected at the same time. Items may be moved by tap-and-drag.

There is a concept of the “current item”, which is the item under the pen. The pen cursor is normally a “pen”, but it changes to a “hand” when it is over an item. The internal identifier of that item, which is defined as the current item, is reported on the top-right of the application’s main window under **Item**. **Label** is the user-supplied label for the current item.

The **Edit** menu supports all the common editing commands including rotation, reflection, adding new elements (this is an alternative to drawing the circuit element), adding/editing parameters, deletion, mass selection, and undo. The **Edit** menu can be invoked by tapping the pen on any open space in the drawing canvas.

Parameters of an item may be added or changed by selecting the item (via a pen-tap on the item) and then selecting **Edit Parameters** from the **Edit** menu. An editing form, shown in Figure 6-2 for an MOS transistor, comes up and the user may use the handwriting recog-

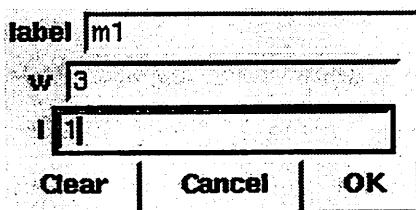


Figure 6-2. Parameter Editing Form for MOS Transistors

nition widget described in Section 4.4.2.1. on page 50 to enter an optional label and the relevant parameters. The form is customized to the current item. For wires, the form contains only a label entry.

The **Speech** menu brings up a widget that controls the speech recognizer described in Section 5.5. on page 87. This widget is shown in Figure 6-3 and allows the user to control

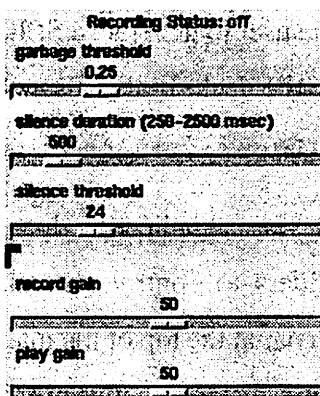


Figure 6-3. Speech Recognition Control Widget

noise immunity, silence detection, and gain.

All commands on the **Edit** menu may be spoken at any time. The list of speakable commands is in Table 6-1. This list is limited to the most commonly used commands to maxi-

Rotate 90
Reflect X-ray
Reflect Yankee
Edit Parameters
Add Resistor
Add Capacitor
Add NMOS
Add PMOS
Add Vdd
Add Ground
Select All
Deselect All
Delete Selection
Undo

Table 6-1. List of Speakable Commands

mize recognition accuracy. Speakable items are kept short for ease of use. An attempt is

also made to differentiate the items as much as possible. That is why we use “Reflect X-ray” and “Reflect Yankee” instead of “Reflect X” and “Reflect Y” respectively.

The **SPICE** menu allows the user to generate a SPICE deck and specify the path and name of the file in which to store the deck. It also allows the user to pull up forms for entering simulation control parameters.

As explained in Section 6.1.3, this application does not use pen double-taps nor the barrel button. There are three sliders to control critical, user-dependent variables. They are located just below the menu bar at the top of the main window. The **Alignment Threshold** slider specifies the maximum distance between the end of a line and the nearest item’s nearest terminal for them to be electrically connected. If they are connected, then the item that was just added or moved is aligned to the existing item.

The **Line Length Threshold** slider specifies the minimum length of a stroke or line segment before it is considered a line. This parameter is used in two cases. The first case is when the pen is tapped. If the line length generated is smaller than the parameter, the event is considered a tap, not a line. In the second case, whenever a stroke is segmented into straight lines, each resulting line’s length is compared to this parameter. If the length is smaller, then it is considered noise and ignored.

The **Recognition Timeout** slider specifies the time after the pen is lifted before recognition begins. If the user finds that recognition occurs before he has finished his strokes, he should increase the value of this parameter. If he finds that recognition begins too late for his comfort, he should decrease the value of the parameter.

All three parameters are initialized with reasonable defaults.

6.1.5. File Format

The file format used by the Circuit Schematic Recognizer is illustrated in Figure 6-4 for the circuit in Figure 6-1, and the syntax of the file is illustrated in Figure 6-5. The first line contains the name of the circuit, some of its parameters, and a list of its elements. This is followed by a line for each circuit element, including wires. Each line contains the type, name, location on the screen, orientation, label, names of connected neighbors for each ter-

```

circuit test 1 5 10 1900 20 r1 c1 v1 n1 p1 g1 wire0 wire1
wire2 wire3 wire4 wire5 wire6 wire7 wire8 wire9 wire10 wire11
wire12 wire13
resistor r1 244 207 0 none 2 wire10 wire11 1 1 1 -1
capacitor c1 202 206 0 none 2 wire8 wire13 2 1 1 -1
Vdd v1 147 111 0 none 1 wire2 1 1 -1
nmos n1 133 206 0 none 3 wire5 wire0 wire1 2 2 1 2 -1 -1
pmos p1 136 151 0 none 3 wire3 wire2 wire0 1 2 1 2 -1 -1
ground g1 139 264 0 none 1 wire1 2 0
wire wire0 2 155 171 152 206 none p1 drain n1 source 0
wire wire1 2 152 226 149 264 none n1 drain g1 ground 0
wire wire2 2 157 121 155 151 none v1 Vdd p1 source 0
wire wire3 3 136 161 103 169 none p1 gate wire4 1 0
wire wire4 2 103 169 107 215 none wire3 2 wire5 1 0
wire wire5 1 107 215 133 216 none wire4 2 n1 gate 0
wire wire6 3 102 192 56 198 none none none none 0
wire wire7 3 232 189 152 189 none wire9 1 none none 0
wire wire8 2 211 191 211 206 none none none c1 positive 0
wire wire9 1 232 189 312 190 none wire7 1 none none 0
wire wire10 0 247 207 250 191 none r1 positive none none 0
wire wire11 2 247 233 249 255 none r1 negative wire12 1 0
wire wire12 3 249 255 152 255 none wire11 2 none none 0
wire wire13 2 211 223 209 252 none c1 negative none none 0

```

Figure 6-4. Circuit Schematic File Example

```

circuit <circuit_name> <scale> <line_threshold>
    <align_threshold> <timeout> <element_count>
    <element_1 element_2...>

<element_type> <element_name> <x_coordinate> <y_coordinate>
    <orientation> <label> <wire_count>
    <wire_name_1 wire_name_2...>
    <wire_terminal_1 wire_terminal_2...>
    <parameter_count> <parameter_1 parameter_2...>

    .
    .
    .
    .

```

Figure 6-5. Circuit Schematic File Syntax

rninal, the names of the connected terminals of each connected neighbor, and all parameters for that element.

6.1.6. Suggested Improvements

The Circuit Schematic Recognizer works well for most users, especially after they have gotten familiar with the templates used for recognition. However there are numerous improvements which would make it work even better.

The most visible improvement would be capture of handwritten labels and parameters on the drawing canvas itself. This would remove the need for a separate form for such data entry and also the need for the stand-alone handwriting recognition widget. An extension to this would be to capture drawn gestures as shorthand for menu commands, similar to keystroke shortcuts in regular computers.

Even if we continued to use the current parameter entry form, it would be nice to write directly into the form rather than into a separate widget application. A tk widget that encapsulates the entire handwriting recognition widget application for inclusion into applications is currently under development (see Section 4.4.2.2. on page 51) and using this widget allows this capability.

A color screen would make the application look nicer and allow other modes of user feedback. For example, labels are currently not displayed on the drawing canvas because it would clutter up the screen. But with a color screen the labels could be displayed in a different color. Selected items or the current item could also be displayed in a different color.

A drag-and-drop palette of circuit elements may be useful, but this usefulness is debatable because dragging and dropping an element across the screen requires a large arm movement. Also, the small screen means that the screen real estate consumed by a palette may be too expensive.

The small screen size also limits the size of the circuit that may be drawn. A larger screen supporting a larger drawing canvas would help. Currently, the canvas does not scroll but adding a scrolling capability is clearly a big win on the InfoPad. Other non-mobile platforms may be able to afford a larger screen.

The application is currently modeless, so the operation executed in response to a user event depends on screen context. For example, tapping the pen on the screen executes a different

command if it is over a circuit item than over white space. However there are advantages to moded behavior and it would be interesting to implement an interface with separate drawing and editing modes.

There is currently no drawing grid on the drawing canvas, and no gravity or snap. Implementing a drawing grid would allow easy manhattanization of the wires, especially if the grid is very coarse. This option should be investigated.

The current circuit element recognizer uses a relatively simple template-based algorithm. A more complex algorithm may achieve better recognition accuracy and promote better user interaction. For example, the present algorithm requires that all drawing be done and the pen lifted before recognition is done. It also requires that lines be drawn in a predefined way. There is no writer adaptation. A more sophisticated algorithm could recognize incrementally and adapt to the user.

Casual tests showed that the speech recognizer works well for some users and badly for others. Adaptive training is needed to enable each user to get better recognition accuracy. The full power of the speech recognizer's interface should be exploited in this application.

Not all SPICE control commands are supported in this application. For completeness, even the more esoteric ones should be supported.

6.2. Electronic Notebook

The Electronic Notebook application was designed as a test of the pen inking capabilities of the InfoPad system infrastructure. In particular, it tests the loop consisting of pen packet collection, transmission from the terminal to the Pen Server through InfoNet, transmission of pen resolution data to the notebook application, generation of X events in the application, processing of X events in the X server, and transmission of bitmaps from the X server to the terminal through InfoNet. This application was crucial in identifying timing bottlenecks early in the project.

The application also highlights the ability of the pen to function as a pointing device as well as a drawing tool.

6.2.1. Desired Functionality

The Electronic Notebook functions as a personal appointment calendar. Users use the pen to navigate through the calendar's pages by clicking on the appropriate date. Once at the desired date, the application displays a writing canvas upon which the user may write or sketch. Handwritten or sketched items may be sent to a recognizer for processing and the resulting representations displayed instead of electronic ink.

There should be no limit to the number of pages per day, nor to the size of each page. There should also be no limit to the complexity of the ink on each page.

6.2.2. Lessons Learned

Developing an electronic ink application was an interesting early learning experience. When drawing on the screen, we found that horizontal ruled lines were essential for maintaining coherency in handwriting. Without lines, users write sloppily and very large, exacerbating the limitations of the small screen. Having reasonably spaced lines improved readability and user satisfaction.

All pen digitizers from the same vendor do not have identical parameters. Each piece has to be calibrated separately. Also, the digitization is not linear. Even if the pen registration is accurate at all four corners, it may not be correct at the center of the screen or, more commonly, along the edges of the screen. Therefore, a compromise was made by registering along the centers of the four edges rather than at the corners.

Existing standards for storing electronic ink are not really standard. The JOT standard [Slat93] had plenty of industry support but was short-lived. The UNIPEN standard [Guyon94] is relatively new and it remains to be seen whether widespread university and research laboratory support ensures its survival.

Feedback to the user is extremely important. When the pen cursor was turned off, users were hesitant and made many mistakes. When buttons did not light up on pen focus, users were again hesitant and made mistakes. This is especially so because of the parallax problem and imprecise pen registration.

6.2.3. Implementation

The Electronic Notebook was implemented using Tcl and the Tk toolkit for the graphical user interface and C to process and store electronic ink. Its interface is shown in Figure 6-

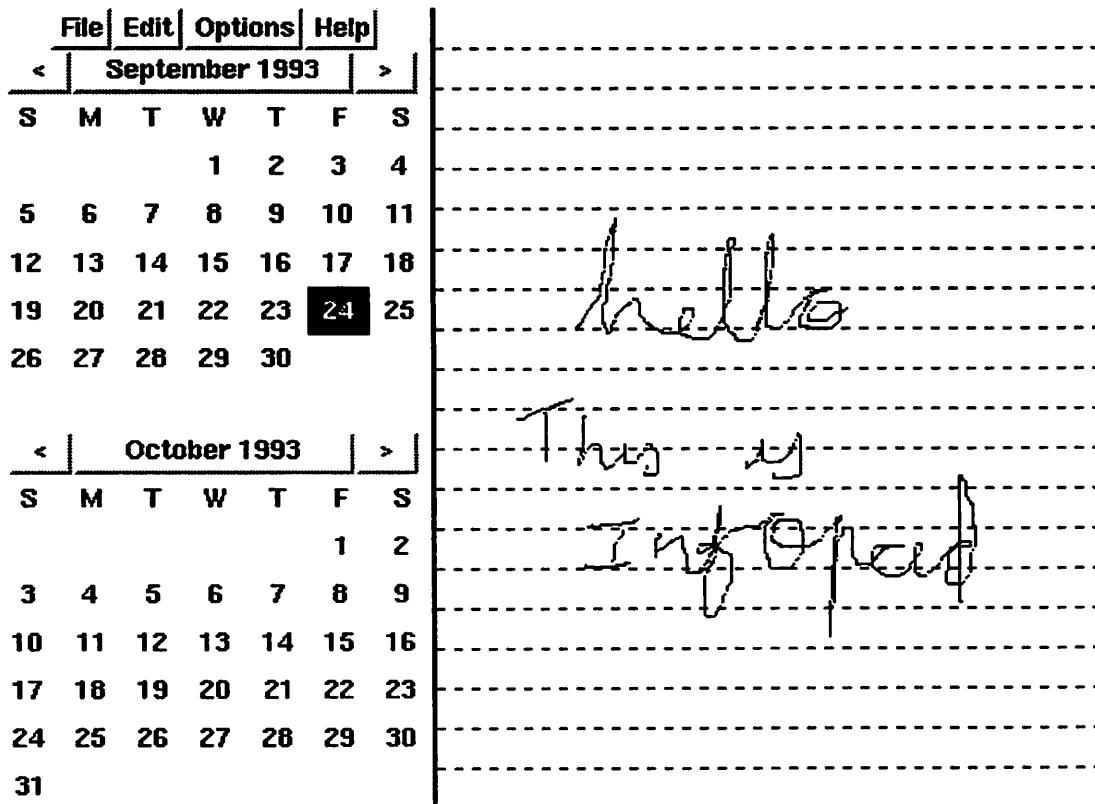


Figure 6-6. Screen Shot of Electronic Notebook Application

6. Users tap the pen on the desired date to pull up the page corresponding to that date. The arrows beside the month allow navigation to other months. The usual file manipulation and editing functions are supported via pull-down menus. The user may sketch or write anything he likes. There is no limit to the complexity of the writing nor to its extent within the page.

The pages are lined to reduce writer sloppiness and to aid future inclusion of handwriting recognition. The line spacing is controlled via the **Options** menu.

Due to the lack of stable standards for storage of electronic ink, we designed our own data storage format and disk format. Internally, **Strokes** are stored as linked lists of pixels. Any piece of writing is stored as a **Word**, which is a linked list of **Strokes**. The **Stroke** and

Word structures are shown in Figure 6-7. The **Baselines** in the **WordStruct** are useful to

```
typedef struct _StrokeStruct{
    int          *x, *y;
    int          PixelCount;
} StrokeStruct;

typedef struct _WordStruct{
    StrokeStruct *Strokes;
    int StrokeCount;
    int UpperBaseline; /* smaller number than
                         * LowerBaseline */
    int          LowerBaseline;
} WordStruct;

typedef struct _TsetDataStruct {
    WordStruct *Words;
    int          WordCount;
} TsetDataStruct;
```

Figure 6-7. Electronic Ink Stroke and Word Structures

a handwriting recognizer. Writing on separate lines are stored are separate **Words**, and all writing for a particular page is stored in a **TsetDataStruct**.

Originally, the **StrokeStruct** had an array of pixels and the **WordStruct** had an array of **Strokes** rather than linked lists. However, it turned out that if the application runs for several days without interruption, sufficient memory fragmentation occurs that the workstation runs out of memory. Using linked lists, the application does its own memory management and therefore does not suffer from memory fragmentation.

The file format for electronic ink is shown in Figure 6-8. **WordCount** is the number of lines of handwriting on a page, **StrokeCount** is the number of strokes on the line. **PixelCount** is the number of pixels in the stroke. The **UpperBaseline** and **LowerBaseline** are the lines bounding the writing area. In this case, the writing area is bounded by the drawn lines above and below the writing. This file format mirrors the internal handwriting representation.

```
WordCount= 1
StrokeCount= 3
UpperBaseline= 50
LowerBaseline= 100
PixelCount= 6
112 680
112 681
112 685
112 688
111 693
136 778
PixelCount= 4
112 685
112 688
111 693
136 778
PixelCount= 4
212 685
212 688
211 693
236 778
```

Figure 6-8. Electronic Ink File Format

6.2.4. Suggested Improvements

The most visible improvement would be to implement scrolling of the writing area. Currently, there is no scrolling.

Currently, handwriting recognition is not connected to the application, so handwritten input must remain electronic ink. It is desirable to allow the user to segment the drawing area and allow him to send the contents of selected areas to a handwriting recognizer. Recognized text could then replace electronic ink. The same applies to drawings.

Current editing capabilities are very basic, comprising selection and deletion. Addition of scaling and translation would help, as well as merging and smoothing of strokes.

7 Conclusions

7.1. Summary of Results

In this thesis, we have reported the design and implementation of a pen and speech based User Interface Architecture for mobile multimedia terminals. Although this architecture is targeted towards the InfoPad system, most of its components are equally usable in other environments. In particular, the author considers Computer Aided Design (CAD) to be the most promising area in which a pen and speech based user interface can add value and succeed.

The pen and speech input paradigm could be the wave of the future in user interfaces for small computers. This interface requires a large amount of computational power to be effective, and this computational power is not currently available to portable computers. The power of this architecture lies in its network-centric design paradigm. By using the resources of a fast network and compute servers, the architecture overcomes many of the limitations of current portable computers that prevent their adoption of recognition-based user interfaces.

The research contributions reported in this thesis include the User Interface Architecture design and the implementation of all of its components. We also examine the situations in which handwriting recognition is preferable to speech recognition, and vice versa. These contributions are summarized in the following sections.

7.1.1. User Interface Architecture

We designed and built a network-based user interface architecture which allows remote provision of pen and audio data services as well as handwriting and speech recognition services to applications. The servers run remotely and communicate with applications and

with each other over the Internet, thereby off-loading the computational demands of compute intensive servers such as recognizers from the application's CPU.

This architecture has several major components. The Pen Server translates a raw byte stream from the InfoPad terminal's pen digitizer into a device-independent format and over-samples this data in space for device-independent spatial resolution. Data is accessed via the Pen Server's API. The Pen Server also emulates the mouse by generating X mouse button and motion events, sub-sampled in time to avoid overwhelming the X server and sub-sampled in space to match the resolution of the screen.

The Audio Server reads an 8 kHz 8-bit µ-law audio stream from the InfoPad terminal's microphone and makes this data available to applications. Applications wishing to access the terminal's speaker must send the data to the Audio Server, which mixes data from all sources, translates it into 8-bit µ-law, and sends it to the terminal. The Audio Server is accessed via the standard AudioFile API so that existing AudioFile-compliant applications do not need modification nor re-compilation to use the InfoPad's audio capabilities.

The handwriting and speech recognizers run as servers on remote machines since they are very compute-intensive. The recognizers are summarized in the following sections.

7.1.2. Handwriting Recognition

In the handwriting recognition portion of the thesis, we analyzed the InfoPad system's handwriting recognition requirements. We concluded that handwriting recognition is best employed in specifying file names, URLs and e-mail addresses. A handprint recognizer is necessary for this purpose. A cursive recognizer is useful for mass text entry but since this is not a primary application for the InfoPad, cursive recognition is optional.

We therefore built a hidden Markov model based handprint recognizer which has two independent vocabularies. The 61-character recognizer supports all uppercase and lowercase alphabetic characters and some special characters. The digit recognizer supports the ten numerals. We also examined the user interaction model, programming model and APIs for access to the recognizer, providing three different access mechanisms.

7.1.3. Speech Recognition

Speech recognition is necessary in the InfoPad system to issue commands. This requires a small, flexible recognizer which can tailor itself to the current application. Such a system was developed and deployed by another student. It does not run as a remote server but is compiled into the application. This recognizer runs fast enough the it does not load down the CPU very much. As in handwriting recognition, we examined the user interaction model and the programming model. Only one access mechanism for this recognizer is currently provided.

For general dictation, a large vocabulary speaker independent system is required. This system requires a large amount of computation. A custom hardware solution using custom VLSI chips was built which could be the basis for a spoken dictation server. This hardware system supports several hidden Markov model based speech recognition algorithms. As speech recognition technology improves, new algorithms may be similarly implemented in a hardware server for real-time performance. This recognizer would be accessed by applications as a remote server.

7.1.4. Applications

We built applications to exercise and test the User Interface Architecture and all of its components. The primary application is the Circuit Schematic Recognizer, a Computer Aided Design (CAD) application that uses pen input for drawing circuit elements and wires, speech recognition for commands, and handwriting recognition for parameters. We discovered many characteristics of using pen digitizers that are different from interacting with the mouse. The use of both the pen and audio input modalities in the same application allowed us greater insight into this new user input paradigm.

7.2. Future Work

There are several areas in which further development could be done and others in which further research is promising. Due to the need to graduate in a finite amount of time, we were not able to explore all these issues in depth as part of this work. We examine the development topics and the research topics as separate categories, since the former would

be an integral part of the InfoPad project but the latter would be applicable in other environments as well and are good starting points for future Masters or Ph. D. projects.

7.2.1. Development Projects

There are several small projects that fall within the scope of the user interface effort which would benefit the InfoPad project. They are development projects which are not generally applicable to other systems although some of the ideas expressed below are certainly transferable.

7.2.1.1. Audio Focus Manager

There is currently no infrastructural control of audio focus. All applications may write and receive audio data at any time, and it is up to the application programmer to provide the user with the capability to turn an audio stream on or off. There is also no mechanism for the application to determine if it has mouse focus.

This scheme is clearly not optimal. The user should be able to control the audio characteristics of all applications in one central place. On the uplink, audio focus should be controllable by the user and the application so that applications may elect to receive audio only on cue, only on pen focus, or continuously. On the downlink to the speaker the overall volume and the relative volume of all audio streams should be user-controllable. The user should also have the option of turning off the microphone and speaker via software control.

The Audio Focus Manager will also allow exploration of other models of access to audio data.

7.2.1.2. Pen Server Control Widget

The Pen Server has several parameters that are either hard-coded, specified on the command-line or specified by the applications that connect to the Pen Server via its API. These parameters should be accessible to the user via a Pen Server Control Widget so that his preferences may be used in the user interface. One such set of parameter is registration information, which should be supported by allowing the user to align the pen whenever he wants by running a registration routine via the Pen Server Control Widget.

Another parameter is the presence or absence of a pen cursor. Some users prefer that there not be a pen cursor. Others prefer that the pen movements not be tracked by the pen cursor when the pen is up or, conversely, when the pen is down. Control of all these behaviors is available to the applications programmer but not to the user.

7.2.1.3. Handwriting Recognition Widget

The handwriting recognition widget needs a lot of work. It needs a raised baseline to allow for characters which have descenders, such as g and y. It needs to talk to all applications rather than just the Tk-enabled ones. Most of all, it needs to be encapsulated into a widget that is included in the application rather than running as a separate application. This would make for a better user interface since users write directly on the input form rather than in a separate window. All widgets connect to a single recognizer similar to the one described in Section 4.7.3.3. on page 65, except that this recognizer will be running as a server rather than compiled into the application. If this were not the case, the application could grow to be unreasonable large if there are many entry boxes on the form.

The current correction mechanisms should be augmented or replaced with drawn gestures that are recognized and interpreted. The current correction mechanism is clumsy and occupies too much screen area, especially if the widget is to be replicated many times, once for each entry box on a form.

7.2.1.4. Speech Recognition Widget

The current speech recognizer has one control widget per application, but many of the parameters are common to all applications, such as garbage threshold, confidence threshold, gain, silence time-out, and noise/silence threshold. All these parameters should be controllable for all applications via just one widget. This widget should also provide user feedback by reporting the most recently recognized utterance.

7.2.1.5. User Interface Control Widget

This widget is an application that provides controllability and observeability of all the components of the User Interface Architecture. The user would use this widget to control the

parameters of each component. In addition to controlling each of the data and recognition servers, it would also allow control of all parameters that are common to many applications, such as gesture interpretations and parameters for the encapsulated handwriting recognition widget. These parameters include tick-mark separation, recognition time-out, writing area size, and handedness (left or right). Handedness information can help recognition accuracy and tell applications on which side of the pen (left or right) to post menus.

7.2.1.6. Other Improvements

There are some other improvements that would help the InfoPad terminal's user interface. Using a Wacom digitizer instead of a Logitech Gazelle would allow use of a smaller pen that does not require batteries. 16-bit, 16-kHz audio in and out would greatly improve recognition accuracy and sound quality. A color screen would improve readability and allow more modes of user feedback. Modifier buttons, as described in Section 1.4.1. on page 7, would allow more modes of user input, as would additional function buttons. Using the screen in portrait mode would make the paper-like interface more complete.

There is currently no formal way to benchmark the system and quantify how well we are doing in terms of the quality of the user interfaces we build. We need to come up with real tests and gather statistics on the user acceptability of the interfaces.

7.2.2. Research Projects

There are some areas of future work arising from this thesis which are applicable to a wide range of pen and speech based systems, and which are good research areas in their own right. Some of these areas are described below.

7.2.2.1. Complex Event Manager

Currently, the InfoPad is cognizant of pen events, audio events, handwritten events and speech events, the latter two being results of recognition. However these events work in isolation. There is no synergy in these events and modes of interaction. Research needs to be done in discovering the kinds of complex, multi-modal events which make sense and in establishing standard interpretations of these events which may be shared across applications.

In order to do this, the Pen and Audio Servers as well as the handwriting and speech recognizers should communicate with applications via a Complex Event Manager. This Manager would look for complex events, which are sets of simple events with special, pre-defined timing and/or spatial relationships, and interpret these events before passing them on to applications. This is analogous to how a window manager such as twm works in the X Window System.

7.2.2.2. Synergy Between Handwriting and Speech Recognition

In the InfoPad system, the handwriting and speech recognizers work independently without taking advantage of the fact that handwriting and speech recognizers tend to have errors that are orthogonal, and can therefore complement each other. At first glance, one method would be to have the user write as well as say all input, using handwriting to generate a list of recognition hypotheses and speech to pick the correct hypothesis from this list. But this is very clumsy. Users balk at having to both write and say everything.

A more realistic approach would be to do post-recognition correction. If the handwriting recognizer makes an error, the speech recognizer could be used to select from the list of hypothesis subsequently: speech is only used if an error is made by the handwriting recognizer.

Another area for exploring the synergy between handwriting and speech recognition would be in complex events, similar to events described Section 7.2.2.1. In this case, the user would use speech as a modifier for handwritten characters or gestures.

7.2.2.3. Handwriting and Speech Recognition

Handwriting and speech recognition are major research areas in their own rights, with many well-organized and capable groups of people working on improving recognition accuracy and reducing computational overhead in terms of processing power and memory requirements. Work is also in progress on recognizers that are robust in the presence of noise and varied data capture hardware. User-adaptive recognition is a hot research area. Progress in all these areas is critical to improving future recognition-based user interfaces but there are other characteristics of recognition that must be addressed as well.

Most current recognizers are not well packaged for use in real applications. More work needs to be done on applications programming interfaces and in encapsulating the programming interface to insulate the programmer as much as possible from the operational details of recognition. The work reported in this thesis goes some way towards attacking this problem but there is a lot more to be done.

Models for user interaction with recognizers are currently primitive. More work needs to be done on encapsulating the user's interaction with recognition objects and on providing feedback to the user on the performance of the recognizer and the recognition results. For example, how does the user know what the speech recognizer detected? DragonDictate [Drag94] uses a control window that displays the recognition results of the latest utterance but surely there are better ways, such as audio cues, which can be application dependent.

7.2.2.4. Integrated Document Editor

The Integrated Document Editor may be the killer application that will show that handwriting and speech recognition has arrived. This application would use cursive handwriting recognition or spoken dictation recognition for mass text entry, printed handwriting recognition for file names and other dictionary words, handwritten gesture recognition or spoken command recognition for keystroke shortcuts or short commands, and drawn geometric object recognition for sketches, computer aided design and other specialized applications.

The challenge in building this application would be integrating the recognizers and drawing areas in such a way that it is possible to determine which recognizer to use for any one input object. This application would also drive the development of the various kinds of recognizers and use the entire user interface infrastructure. A related challenge is the user feedback and data display problem. This is the issue of letting the user know that his input data and commands have been accepted and correctly interpreted, and the related issue of recovery from recognition errors.

This application would allow exploration of all the issues examined in this thesis and, if successful, would be a harbinger of other recognition-based applications to come.

8 Appendix: Handwritten Character Sets

In this appendix we list the various ways of writing each character in the training set used for handwriting recognition. The characters in the 61-character recognizer are in Table 8-1 and the digits are in Table 8-2. For multi-stroke characters, the strokes can often be written in any order. The arrowhead at the tip of each stroke indicates the direction of the stroke.

Character	Representations
a	
b	
c	
d	
e	
f	
g	
h	
i	
j	
k	
l	
m	

Table 8-1. Character Set for 61-Character Recognizer

Character	Representations
n	n n
o	o o o
p	p p p p
q	q q q
r	r r
s	s s
t	t t t t
u	u
v	v
w	w w
x	x x x x
y	y y y
z	z z z

Table 8-1. Character Set for 61-Character Recognizer

Character	Representations
A	A A A A A
B	B B B
C	C
D	D D
E	E E E E E
F	F F F
G	G G G
H	H H H H H
I	I I I I I
J	J J J J J
K	K K K K K
L	L L
M	m m

Table 8-1. Character Set for 61-Character Recognizer

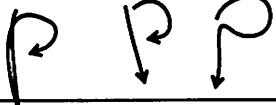
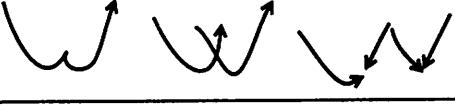
Character	Representations
N	
O	
P	
Q	
R	
S	
T	
U	
V	
W	
X	
Y	

Table 8-1. Character Set for 61-Character Recognizer

Character	Representations
Z	
* (asterisk)	
@ (at sign)	
! (exclamation mark)	
- (minus)	
. (period)	
/ (slash)	
~ (tilde)	
tt (double t)	
_ (underscore)	

Table 8-1. Character Set for 61-Character Recognizer

Digit	Representations
0	○
1	↓ ↓ ↗
2	↙ ↙ ↙ ↘
3	↪
4	↖ ↗ ↗
5	↖ ↗ ↗
6	↙ ↗ ↗
7	↓ ↗ ↗ ↗
8	↑ ↗ ↗ ↗ ↗ ↗ ↗
9	↓ ↗ ↗ ↗

Table 8-2. Character Set for Digit Recognizer

9 Appendix: Software Organization

In this appendix we list the files comprising each piece of software reported in this thesis. All files may be found in the InfoPad software distribution.

9.1. Pen Server

The Pen Server source tree may be found in `/tools/infonet/type_servers/pen/gazelle/vcurrent`, and its component files are listed in Table 9-1.

9.2. Audio Server

The Audio Server source tree may be found in `/tools/infonet/type_servers/audio/vcurrent`, and its component files are listed in Table 9-2. Since the source tree is a modified version of the AudioFile source tree, only files which differ from AudioFile are listed in the Table.

9.3. Handwriting Recognizer

The Handwriting Recognizer source tree is stored in four directories in `/tools/ui/handwriting`. **HMM/vcurrent** contains the source tree for the recognizer itself, including the hidden Markov model code. **HW/vcurrent** contains the source tree of the Data Capture and Manipulation Package and support routines for electronic ink handling and feature extraction. **data** contains the training and test data captured. **models** contains the hidden Markov model parameter files. These directories are listed in Table 9-3 to Table 9-6.

9.4. Circuit Schematic Recognizer

The Circuit Schematic Recognizer source tree is stored in `/tools/ui/schematic/vcurrent`. Its component files are listed in Table 9-7.

File Name	Description
README	Pen Server modification history.
bin/solaris/gazelle_gw	Pen Server executable, solaris version.
bin/solaris/pentest	Pen Server test program, solaris version.
bin/sunos/gazelle_gw	Pen Server executable, sunos version.
bin/sunos/pentest	Pen Server test program, sunos version.
include/gazelle_gw.h	Header file containing function prototypes and data type definitions internal to the Pen Server.
include/penpriv.h	Header file containing data type definitions common to the Pen Server and its API routines.
include/penproto.h	Header file containing function prototypes and data type definitions for the API. This file is included by applications that require pen-resolution data.
lib/solaris/libpenlib.a	Object library file for Pen Server API, solaris version.
lib/solaris/libpenlib.so	Object library file for Pen Server API, solaris version.
lib/sunos/libpenlib.a	Shared object library file for Pen Server API, sunos version.
lib/sunos/libpenlib.so	Shared object library file for Pen Server API, sunos version.
man/man1/gazelle_gw.1	Manual page for the Pen Server.
man/man3/penlib.3	Manual page for the Pen Server API.
scripts/penns	Script to start up the Pen Server with the appropriate command-line options and UNIX environment variables.
scripts/penns.tcl	tcl script to start the Pen Server with the appropriate command line variables. penns.tcl also registers the Pen Server with the InfoNet Name Server.
src/gazelle_gw.c	C source file for the Pen Server.
src/penlib.c	C source file for the Pen Server API.
src/pentest.c	C source file for the Pen Server test program.

Table 9-1. Pen Server Source Tree

9.5. Notebook Application

The Notebook source tree is stored in /tools/ui/apps/notebook/vcurrent. Its component files are listed in Table 9-8.

File Name	Description
AF/server/dda/sparc1/sparc.c	Code for the part of the Audio Server which connects to the InfoNet Pad Server to obtain audio uplink data and to send audio downlink data.
bin/solaris/Asparc1	Audio Server executable, solaris version.
bin/sunos/Asparc1	Audio Server executable, sunos version.
doc/user_guide	Document describing how to run the Audio Server and how to use its API.
scripts/audions	Script to start up the Audio Server with the appropriate command-line options and UNIX environment variables.
scripts/audions.tcl	tcl script to start the Audio Server with the appropriate command line variables. audions.tcl also registers the Audio Server with the InfoNet Name Server.

Table 9-2. Audio Server Source Tree

File Name	Description
lib/solaris/HMM.a	Object library file containing the entire recognizer with the Sun API.
lib/solaris/HMM.so	Shared object library file containing the entire recognizer with the Sun API.
man/man1/recogCursive.1	Manual page for the stand-alone handwriting recognition server.
man/man3/recoglib.3	Manual page for the API of the stand-alone handwriting recognition server.
src/Edge.{h,cc}	Source and header files for Edge objects in the recognizer. These objects contain information relating to transitions between pairs of states.
src/HidMarkMod.{h,cc}	Source and header files for HMM objects in the recognizer. These objects contain information relating entire HMMs.
src/LList.{h,cc}	Source and header files for linked list objects in the recognizer.
src/NBest.{h,cc}	Source and header files for NBest objects in the recognizer. These objects contain the N best alternatives returned by the recognizer.
src/PDF.{h,cc}	Source and header files for PDF objects in the recognizer. These objects contain probability density functions.
src/Prob.{h,cc}	Source and header files for Edge objects in the recognizer. These objects contain individual probabilities.
src/SpeechData.{h,cc}	Source and header files for SpeechData objects in the recognizer. These objects contain feature vectors extracted from the input.
src/State.{h,cc}	Source and header files for State objects in the recognizer. These objects contain information relating to Markov states within a HMM, including grammar nodes.
src/StringList.{h,cc}	Source and header files for StringList objects in the recognizer. These objects contain lists of strings, and are used to return recognized results.
src/StringUtil.{h,cc}	Source and header files for utilities that operate on StringList objects in the recognizer.
src/cursive2HmmName.cc	Routines for translating file names to and from the character string represented by that file name.
src/cursiveHMMNames.{h,cc}	Routines for translating file names to and from the character string represented by that file name.
src/hre.{h,cc}	Top level source file for the recognizer with the Sun API.
src/newFileIO.{h,cc}	Routines for file handling.
src/newGiveGrade.{h,cc}	Routines for scoring recognition results versus expected results.
src/recogCursive.cc	Top level source file for stand-alone recognizer.
src/trainCursive.cc	Top level source file for recognizer training.

Table 9-3. Handwriting Recognizer Source Tree

File Name	Description
bin/GetComment.tcl	tcl file containing entry widget allowing users to enter a comment into a data file.
bin/GetDataWord.tcl	tcl file containing entry widget allowing users to specify the word being stored in a data file.
bin/GetFeatureWord.tcl	tcl file containing entry widget allowing users to specify the word being stored in a feature file.
bin/GetWordCount.tcl	tcl file containing entry widget allowing users to specify the number of words to capture in a data entry session.
bin/HW.tcl	main tcl file for the Data Capture and Manipulation Package.
bin/HWtcl	executable of interpreter for tcl files in the Data Capture and Manipulation Package.
include/HW.h	header file for applications that use the Data Capture and Manipulation Library.
include/HWlib.h	same as HW.h.
lib/libHW.a	library file containing all the routines in the Data Capture and Manipulation Package excluding user interface routines.
src/HWPen.h	header file for HWPen.c.
src/HWtcl.h	header file for all routines that use the graphical user interface.
src/HW.c	top level source file for bin/HWtcl.
src/HWAppInit.c	source file for tcl initialization routines.
src/HWCanvas.c	source file for routines that display and manipulate electronic ink on a canvas window.
src/HWCompare.c	source file for routines that compare electronic ink.
src/HWCopy.c	source file for routines that copy electronic ink into a new data structure.
src/HWExtract.c	source file for routines that perform feature extraction.
src/HWFiles.c	source file for routines for file manipulation.
src/HWMemory.c	source file for memory management routines: this package does all its own memory management.

Table 9-4. Data Capture and Manipulation Package Source Tree

File Name	Description
src/HWMouse.c	source file for routines supporting electronic ink capture via a mouse.
src/HWPen.c	source file for routines supporting electronic ink capture via a pen digitizer.
src/HWPrint.c	source file for routines that print information regarding any piece of electronic ink.
src/HWResample.c	source file for routines that re-sample electronic ink.
src/HWSegment.c	source file for routines that segment electronic ink.
src/HWSort.c	source file for routines that sort electronic ink.
src/HWStrokeNeighbours.c	source file for routines that determine which strokes are neighbors: used in segmentation.
src/HWTruncate.c	source file for routines that truncate the last few pixels from any piece of electronic ink.
src/HWWindows.c	source file for routines that display electronic ink and features.
src/Utils.c	source file for general utility routines.

Table 9-4. Data Capture and Manipulation Package Source Tree

File Name	Description
test/digits	test data for digit recognizer.
train/digits1/hwdata	raw captured handwritten digits for the digit recognizer.
train/digits1/segmented	segmented data from train/digits1/hwdata.
train/digits1/vectors	feature vectors extracted from train/digits1/segmented.
train/letters1/hwdata	raw captured handwritten characters for the 61-character recognizer.
train/letters1/segmented	segmented data from train/letters1/hwdata.
train/letters1/vectors	feature vectors extracted from train/letters1/segmented using baseline normalization, sorting, and ResampleRatio = 0.07.
train/letters1/vectors.new	feature vectors extracted from train/letters1/segmented using baseline normalization and ResampleRatio = 0.07.
train/letters1/vectors.test	feature vectors extracted from train/letters1/segmented using sorting, without baseline normalization and re-sampling.
train/letters1/vectors_bl_0_sort	feature vectors extracted from train/letters1/segmented using sorting and baseline normalization, without re-sampling.

Table 9-5. Handwritten Data Source Tree

File Name	Description
macro/digits/digitRecognizer.hmm	HMM file for the digit recognizer.
macro/gen/bin/genTopLevel	executable file to create top level HMM files.
macro/gen/src/genTopLevel.cc	source file for genTopLevel.
macro/letters/169charRecognizer.hmm	HMM file for an old version of recognizer: recognizes 169 distinct characters.
macro/letters/169scharRecognizer.hmm	updated version of 169charRecognizer.hmm.
macro/letters1/printRecognizer.hmm	HMM file for the 61-character recognizer.
train/iter/*.hmm	HMM files for each digit.
train/iter.new/*.hmm	same as train/iter/*.hmm but with probabilities in scientific notation.
train/letters1/4biter_mm_0_sort.new/*.hmm	HMM files for each character in the 61-character recognizer.

Table 9-6. Hidden Markov Model Parameters Source Tree

File Name	Description
bin/schematic	UNIX shell script that forms the executable of the Circuit Schematic Recognizer.
lib/*.xbm	bitmap files for circuit elements.
man/man1/schematic.1	manual page for the application.
src/Vdd.tcl	source file for Vdd class.
src/capacitor.tcl	source file for capacitor class.
src/circuit.tcl	source file for circuit class.
src/classes.tcl	file containing list of circuit element class source files.
src/editmenu.tcl	source file containing all code associated with the Edit Menu.
src/element.tcl	source file for element class, from which all circuit element classes and the wire class inherit.
src/ground.tcl	source file for ground class.
src/line.tcl	source file for line class.
src/nmos.tcl	source file for nmos class.
src/pmos.tcl	source file for pmos class.
src/resistor.tcl	source file for resistor class.
src/schematic.tcl	top level tcl source file for the application.
src/segment.tcl	source file for segment class.
src/spice.tcl	source file containing all code associated with the SPICE Menu.
src/src.tcl	source file for source class.
src/wire.tcl	source file for wire class.

Table 9-7. Circuit Schematic Recognizer Source Tree

File Name	Description
CreateMonthCalendar.tcl	source file for code to create a window displaying the calendar for a given month.
GetDayOfWeek.tcl	source file for calculating the day of the week given the date.
GetDaysInMonth.tcl	source file for calculating the number of days in a given month.
GetNewNotebookName.tcl	source file containing code to prompt user for the name of a new notebook and then read that notebook.
GoToDate.tcl	source file containing code to prompt user for a date and then go to that date.
InitArrays.tcl	source file declaring and assigning global arrays.
SaveQuery.tcl	source file containing code to prompt user to save a modified page before going to another page.
UpdateMonthCalendar.tcl	source file containing code to update the calendar window to display the current month.
notebook.tcl	top level source file for this application.
*.c	source files for electronic ink manipulation, similar or identical to the file in the Data Capture and Manipulation Package, as listed in Table 9-4.

Table 9-8. Notebook Source Tree

Bibliography

- [Aust90] S. Austin, P. Peterson, P. Placeway, R. Schwartz, and J. Vandergrift. “Toward a Real-Time Spoken Language System Using Commercial Hardware.” In *Proceedings of the DARPA Speech and Natural Language Workshop*, pages 72–77, Hidden Valley, PA, June 1990.
- [Baum72] L. E. Baum. “An Inequality and Associated Maximization Technique in Statistical Estimation for Probabilistic Functions of Markov Processes.” In Oved Shisha, editor, *Inequalities III*, pages 1–8, New York, NY, 1972. Academic Press.
- [Burs96] A. Burstein. *Speech Recognition for Portable Multimedia Terminals*. PhD thesis, University of California at Berkeley, 1996.
- [Carr91] R. Carr. “Handwriting Recognition in the GO Operating System.” In *Proc. IEEE Compcon Spring ’91*, pages 483–486, February 1991.
- [Chan93] A. Chandrakasan, T. Burd, A. Burstein, S. Narayanaswamy, Sheng S., and R. Brodersen. “System Design of a Multimedia I/O Terminal.” In *VLSI Signal Processing VI*, pages 57–65. IEEE Press, 1993.
- [Chan94] A. Chandrakasan. *Low Power Digital CMOS Design*. PhD thesis, University of California at Berkeley, 1994.
- [Cran93] H. Crane and D. Rtishev. “Pen and Voice Unite.” *Byte Magazine*, pages 98–102, Oct 1993.
- [Cros95] S. Crosby. “The a2x FAQ.” <http://www.cl.cam.ac.uk/users/sac/a2x-faq.html>, 1995.
- [Doer96] R. Doering, T. Truman, and R. Brodersen. “A Modular Design for Wireless Multimedia Access.” *Kluwer Journal of VLSI Signal Processing*, to be published 1996.
- [Drag94] Dragon Systems, Inc. *Dragon Dictate User’s Guide*. Dragon Systems, Inc, 1994.

- [ERL88] University of California at Berkeley Electronics Research Laboratory. “*LagerIV Distribution 1.0 Silicon Assembly System Manual*,” 1988.
- [Fuji93] T. Fujisaki, K. Nathan, W. Cho, and H. Beigi. “On-line Unconstrained Handwriting Recognition by a Probabilistic Method.” In *Proceedings of the International Workshop on Frontiers in Handwriting Recognition*, pages 235–241, Buffalo, NY, May 1993.
- [Gill95] D. Gillespie and J. Lazzaro. “The Log System.” <http://www.pcmp.caltech.edu:80/chipmunk/>, 1995.
- [Gold91] D. Goldberg and A. Goodisman. “Stylus User Interfaces for Manipulating Text.” In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 127–135, Salem, MA, 1991. ACM Press.
- [Gold93] D. Goldberg and C. Richardson. “Touch-Typing with a Stylus.” In *INTERCHI ’93 Conference Proceedings*, pages 80–87, Salem, MA, April 1993. ACM Press.
- [Guyo92] I. Guyon, D. Henderson, P. Albrecht, Y. LeCun, and J. Denker. “Writer Independent and Writer Adaptive Neural Network for On-Line Character Recognition.” *From Pixels to Features III*, pages 493–506, 1992.
- [Guyo94] I. Guyon. “UNIPEN 1.0 Format Definition.” <http://www.nici.kun.nl/unipen/unipen.def>, 1994.
- [Hass95] M. Hassoun. “SCAPP 9.0 Users Manual.” <ftp://vlsi1.ee.iastate.edu/pub/scapp>, 1995.
- [itc96] “Object-Oriented Programming in Tcl/Tk.” <http://www.wn.com/biz/itcl/>, 1996.
- [Juan84] Juang, B. H. “On the Hidden Markov Model and Dynamic Time Warping for Speech Recognition - A Unified View.” *AT&T B.L.T.J.*, 63(7):1213–1243, January 1984.
- [Kemp93] J. Kempf. “Integrating Handwriting Recognition into Unix.” In *Proceedings of the Summer 1993 USENIX Conference*, pages 187–204, 1993.
- [Kurt94] G. Kurtenbach and B. Buxton. “User Learning and Performance with Marking Menus.” In *CHI ’94 Conference Proceedings*, pages 258–264, Salem, MA, April 1994. ACM Press.
- [Le95] M. Le, F. Burghardt, S. Seshan, and J. Rabaey. “InfoNet: the Networking Infrastructure of InfoPad.” In *Proc. Compcon ’95*, San Francisco, CA, 1995.
- [Lee89] K. F. Lee. *Automatic Speech Recognition*. Kluwer Academic Publishers, Norwell, MA, 1989.

- [Leve93] T. Levergood, A. Payne, J. Gettys, G. Treese, and L. Stewart. "AudioFile: A Network-Transparent System for Distributed Audio Applications." In *Proceedings of the Summer 1993 USENIX Conference*, June 1993.
- [Lyon96] R. Lyon and L. Yaeger. "On-Line Hand-Printing Recognition with Neural Networks." In *Proceedings of the Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, Lausanne, Switzerland, February 1996. IEEE Computer Society Press.
- [MBo95] "MBONE Information Web." <http://www.best.com/prince/techinfo/mbone.html>, 1995. MBone WWW Page.
- [McMu92] L. McMurchie and C. Ebeling. "Wirec 3.2 Tutorial and Reference Manual." <ftp://shrimp.cs.washington.edu/vlsi/wirec.3.2.tar.Z>, 1992.
- [Meta91] Meta-Software, Inc. *HSPICE User's Manual*. Meta-Software, Inc, 1991.
- [Micr95] Microsoft Corporation. *Programmer's Guide to Pen Services*. Microsoft Press, Redmond, WA, 1995.
- [Murv89] Murveit, H. et al. "SRI's DECIPHER System." In *Proc. of the Speech and Natural Language Workshop*, pages 238–242, Feb 1989.
- [Nara96] Narayanaswamy, S. et al. "Application and Network Support for InfoPad." *IEEE Personal Communications Magazine*, to be published March 1996.
- [Oust94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, Reading, MA, 1994.
- [Pall92] Pallett, D. et al. "DARPA February 1992 ATIS Benchmark Test Results." In *Proceedings of the Speech and Natural Language Workshop*, pages 15–27, San Mateo, CA, February 1992. Morgan Kaufmann Publishers.
- [Par96] "Paragraph Handwriting Recognition Technology: Calligrapher." <http://www.paragraph.com/calligrapher/>, 1996.
- [Pen 95] Pen Computing Magazine. "PDA Buyer's Guide." *Pen Computing Magazine*, Aug/Sept 1995.
- [Raba88] J. Rabaey, A. Stoelzle, D. Chen, S. Narayanaswamy, R. Brodersen, H. Murveit, and A. Santos. "A Large Vocabulary Real Time Continuous Speech Recognition System." In *VLSI Signal Processing III*, pages 61–74, New York, NY, 1988. IEEE Press.
- [Rhyn91] J. Rhyne, D. Chow, and M. Sacks. "Enhancing the X-window System." *Dr. Dobb's Journal*, pages 30–38, December 1991.

- [Rhyn93] J. Rhyne and C. Wolf. "Recognition-based User Interfaces." *Advances in Human Computer Interaction*, 4:191–250, 1993.
- [Rowe92] L.A. Rowe and B.C. Smith. "A Continuous Media Player." In *Proc. 3rd Int. Workshop on Network and Operating System Support for Digital Audio and Video*, Nov 1992.
- [RR90] Racal-Redac. "Visula User's Manual," 1990.
- [Sche91] W. Scheifler and J. Gettys. *X Window System*. Digital Press, Bedford, MA, 1991.
- [Schm90] C. Schmandt, M. Ackerman, and D. Hindus. "Augmenting a Window System with Speech Input." *IEEE Computer Magazine*, pages 50–56, August 1990.
- [Slat93] Slate Corporation. "JOT Version 1.0," 1993.
- [Stan86] Stanford University VLSI/CAD Group. "THOR Release 3.2 User's Manual," 1986.
- [Stol92] A. Stolzle. *A Real Time Large Vocabulary Speech Recognition System*. PhD thesis, University of California at Berkeley, 1992.
- [Tapp90] C. Tappert, C. Suen, and T. Wakahara. "The State of the Art in On-Line Handwriting Recognition." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(8):787–808, August 1990.