INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
PENWARE PANTHER: AN EMBEDDED

COMPUTER SYSTEM FOR

REAL-TIME APPLICATIONS

by

Chenyong Huang

Bachelor of Science
Liaoning University of Industry, P. R. China
1986

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree
Department of Electrical and Computer Engineering
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
May 1999
The Thesis prepared by

Chenyong Huang

Entitled

Penware Panther: An Embedded Computer System for Real-time Applications

is approved in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative
ABSTRACT

Penware Panther: An Embedded Computer System for Real-time Application

By

Chenyong Huang

Department of Electrical and Computer Engineering
University of Nevada, Las Vegas

Embedded computer systems target on different tasks and serve in various environments. This thesis relates to the design of Panther, an embedded computer system for real-time applications. Panther, a product of Mobinetix System Company, is a transaction and signature capture terminal which is aimed at a paperless environment. This product possesses a variety of functions from electronically capturing signature for receipts, contracts or identification, to touch-screen communication for PIN entry, advertising and customer survey. The emphasis of the thesis is on design of the firmware module of Panther. Instead of using the traditional flow chart and state machine approaches, an Object Orientation (OO) modeling approach is taken, which improves problem domain abstraction as well as system’s stability in the presence of changes. The Unified Modeling Language (UML) is used throughout the design to express the constructs and relationships among them. This work contributes, mostly to the Panther firmware architecture design, firmware implementation of communication processor, interpreter, and command applications.
# TABLE OF CONTENTS

ABSTRACT ........................................................................................................... iii

TABLE OF CONTENTS .......................................................................................... iv

LIST OF FIGURES ................................................................................................... vii

ACKNOWLEDGMENTS ............................................................................................ viii

CHAPTER 1 INTRODUCTION .................................................................................. 1
1.1 In Real-Time Computer Systems ................................................................. 1
1.2 Characteristics of Real-Time Embedded Computer Systems ..................... 2
1.3 Buses and Real-time Operating Systems ...................................................... 4
1.4 Contributions and Organization ................................................................... 9

CHAPTER 2 PATHER SYSTEM OVERVIEW ......................................................... 11
2.1 System Architecture ..................................................................................... 12
2.2 Memory Map And I/O map .......................................................................... 15
2.3 Panther Interrupt .......................................................................................... 15

CHAPTER 3 FIRMWARE DESIGN ........................................................................... 19
3.1 System Analysis ........................................................................................... 19
3.2 Using OS-9 as the Operating System .......................................................... 22
3.3 Introducing UML ........................................................................................ 24
3.4 Command Definition .................................................................................. 26
3.5 Communication protocol definition ............................................................. 29
3.5.1 Packed Data Format ............................................................................. 29
3.5.2 Header Description .............................................................................. 29
3.5.3 Host Message Description .................................................................. 30
3.5.4 Pad Return Message Description ......................................................... 31
3.6 Firmware Architecture ............................................................................... 32
3.6.1 Communication Processor .................................................................. 32
3.6.2 Panther Interpreter .............................................................................. 33
3.6.3 Command Applications .................................................................... 34
3.6.4 PAL .................................................................................................... 35
3.6.5 Interconnection .................................................................................... 42

CHAPTER 4 FIRMWARE IMPLEMENTATION ..................................................... 45
4.1 Thread Assignments .................................................................................... 45
4.1.1 Communication Thread ...................................................................... 45
4.1.2 Communication Sending Thread ........................................................ 45
LIST OF FIGURES

Figure 2.1 Vadem VG230 Block Diagram.................................................................13
Figure 2.2 Panther Memory Map.............................................................................16
Figure 2.3 Panther I/O Map ..................................................................................17
Figure 2.4 Panther Interrupt Configuration............................................................18
Figure 3.1 Firmware blocks diagram......................................................................21
Figure 3.2 Primary Diagrammatic Elements............................................................32
Figure 3.3 Communication Processor Package Diagram.........................................33
Figure 3.4 Panther Interpreter Package ..................................................................34
Figure 3.5 Command Applications Package...........................................................35
Figure 3.6 PAL package..........................................................................................36
Figure 3.7 Touch-Pad Function State Model............................................................38
Figure 3.8 MSR Function State Model....................................................................39
Figure 3.9 Panther Firmware Layered Architecture pattern......................................44
Figure 4.1 Communication Receiving Thread State model.......................................46
Figure 4.2 Communication Sending Thread State Model.........................................47
Figure 4.3 Dispatching Thread State Model.............................................................49
CHAPTER 1

INTRODUCTION

In this chapter, we first introduce real-time embedded systems and describe their main characteristics, which are deterministic response and minimal latency. Following that, we introduce the key elements in a real-time embedded system, e.g. bus and real-time operating system. The last section of this chapter presents the contributions and organization of the thesis.

1.1 Introducing Real-Time Computer Systems

A real-time embedded computer system is the one that accepts some type of input from its environment and generates output for that environment in an immediate and predictable way. In other words, real-time embedded computer systems operate in the here-and-now, interacting with the real world on a microsecond-by-microsecond basis. In the book "Introduction to Real-time Software Design"[1], Allworth describes the task of computers in the real-time embedded systems as being engaged in interacting with physical devices and processes whose dynamics follow their own laws in a system-wide physical time frame.

Real-time embedded computer systems exist everywhere, from children's toys to wash machine and automobile. They are even more prevalent in industry. Switching systems,
aircraft, nuclear power plants all use the real-time embedded computer systems. Actually, real-time embedded system improves our quality of life safely and conveniently.

Real-time computer systems began to appear early on in the history of computers, but until the advent of the microprocessor in the early -70's, these systems were large and cumbersome ones built around at least a mini-computer and sometimes even a mainframe machine. They were also highly expensive. The microprocessor changed all that, making it feasible to embed complex control functions in devices as mundane as washing machines or as exotic as the Mars Rover. To better understand real-time embedded systems, we present the main features of the systems in the next section.

1.2 Characteristics of Real-Time Embedded Computer Systems

There are two primary characteristics [12] that distinguish a real-time embedded computer system from their more visible desktop cousins.

(1) Real-time systems respond to events in the real world in a timely and predictable fashion.

(2) A real-time system operates as quickly as possible to reduce the amount of time between input and output, and that is the reason of the term: "real-time".

In contrast, a typical modern PC may seem rather fast, but in reality its response to a user typing at the keyboard can vary over a wide range of time. The variance or the lack of a timely response is one of the factor that make it difficult, or even impossible to use something like Windows or Unix as a real-time system.
Time is of the essence in real-time embedded computer systems. Although, the time under discussion here is small, on the order of tens of milliseconds, when one is dealing with real-time events, a millisecond can sometimes make the difference between success and failure. Even worse, multi-processing operating systems such as Windows/NT or Unix may occasionally suspend some processes in order to service a disk drive or handle a network connection. The time required for a system to respond to an event is called "latency". The lower the latency the better the system will respond to events which require immediate attention. A high latency level will make a system to deal with an event after a period of time when the event no longer has any meaning. This type of latency is not acceptable in a real-time situation.

It is not enough that a real-time system is able to exhibit low latency, it must also be predictable [12]. We know that a system that is non-deterministic cannot be predicted. This kind of system may respond immediately to an external event, but then it takes a random amount of time to process the event and resumes whatever other tasks are currently ongoing. A typically general-purpose operating system is non-deterministic, and may exhibit period of high latency [6].

Non-deterministic response is not suitable for a real-time system. The reason is that when designing the software for a real-time system, the programmer needs to know how long it will take to perform a particular action so that the time intervals inherent in the software can be accounted for in the overall system design.

Through above analysis, we conclude that a real-time system requires both deterministic response and minimal latency. In addition, a real-time system is often
implemented in a highly constrained environment, such as a small microprocessor with a minimal amount of memory to work with.

How does a real-time embedded computer systems connect with the outside world? Generally this task is done by signals, also known as interrupts, which inform the processor when something needs attention. In a real-time system, it is up to the software to handle the interrupt, receive or send the required data, and then resume system processing as quickly as possible, within a specific period of time.

A real-time system may also periodically check, or poll, an input signal to watch for changes. This is trickier from a latency standpoint, since the programmer has to predict in advance when the signal will be polled to ensure that it happens often enough and at the right times.

Once input has been processed, the system will typically need to do something about it. The urgency of the response is preset by assigning it a priority in the software. This helps to ensure that events which are considered high-priority be processed before events which have a lower, and hence less critical, priority. The response priorities must be designed to fit within system latency constraints and overall throughput requirements.

1.3 Buses and Real-time Operating Systems

In this section, we present the important elements in the real-time embedded computer systems. There are two key elements in a real-time embedded system, which mostly decide the performance of the system, the cost, the design circle, and the system maintenance. The two key elements: one is bus and another is operating system. By comparing the different types of the buses and operating systems, we can get insights into the design
decision we made for our real-time embedded system, Father, which is illustrated in the following chapters.

The type selection of the bus is critical. It determines how much addressing range is available, what hardware interrupts are available, and ultimately what type of software will be used to operate the finished system. In addition, each bus type has its own particular strength and weakness in terms of performance and physical endurance. All of these factors must be considered when selecting new hardware.

There are many different types of system bus available today, such as VME (Versa Module Europe), PC104, Multi-Bus, ISA (Industry Standard Architecture) [17], and PCI (Peripheral Component Interconnect) [16]. The three most commonly encountered in real-time systems are VME, ISA, and PCI. In some cases a custom bus is developed to meet project objectives.

ISA is the original PC/AT type bus that first appeared sometime around 1984. The 8-bit (or XT) version only supports a limited number of discrete I/O addresses, 8-bit data transfers, and one of four hardware interrupts. Some I/O cards, such as digital interface and low-end timers still use the 8-bit ISA interface. The full AT version supports 16-bit data, an extended range of I/O addresses, and allows full access to almost all of the PC's hardware interrupts.

PCI bus is faster than ISA and may replace ISA soon as the primary bus in PC class computer systems. PCI bus supports bus mastering (a PCI card can take control of the system for data transfers and such), and it is 32-bits wide. However, some manufacturers have decided to take full advantage of the 32-bit capabilities of PCI, and have designed their products so that data is transferred in memory address regions above the old 1-
megabyte DOS limit. This means that if one has an existing science package written in a programming language which is not 32-bit compatible then one will have to rewrite all of one's software for 32-bit operation. There is really no easy way around this.

VME is an established industry standard, and is commonly found in mission-critical applications where reliability and ease of programming are major concerns. VME supports 32 and 64-bit modes, multiple interrupts and bus mastering. In many ways, VME is similar in terms of capabilities and performance to PCI, but its physical design is much more robust.

Operating system is another key element for real-time embedded system. The small embedded systems with simple operations may not necessarily need operating systems. As embedded systems become large and complicated, using real-time operating systems, or at least real-time kernels, becomes very necessary. The basic rule is that what is not in the operating system may end up in the application. Operating systems simply make software much easier to write and debug.

It is not that any operating system is appropriate for the real-time embedded systems [6]. To meet the characteristics, the operating systems used in the real-time embedded systems should possess the properties of multitasking, inter-task communication mechanism, task scheduling, error handling, reliable memory protection, large memory range, networking, and acceptable system latency.

Now, let us simply analyze some of the popular operating systems for desktop computer to see if they are good candidates for real-time embedded computer systems.

One of the disadvantages of DOS [2] is lack of multitasking. The low-level BIOS software loaded by DOS at boot time contains code which is not reentrant, meaning that if
one did run more than a single process at a time, the multiple processes could not share the same low-level DOS functions without causing a serious system crash. There are real-time versions of DOS available, some of which are quite powerful. There are also some multitasking DOS variants. DOS lacks an efficient error handling facilities. Someone will have to design and incorporate these features if they are required. In addition, DOS does not have good memory protection. A program can reach any part of memory or any I/O address. If the program has a bug, this can be a total disaster.

In MS-DOS, one must resort to device drivers and TSR (terminate-and-stay-resident) functions to achieve network capability. Because of limited memory space, after installing network software, there may not be enough space left to run any serious application. The lack of multitasking in DOS also means that one cannot have a network operation active concurrently with other application, unless the network I/O is an intimate part of the application program.

The modern free Unix-like operating systems such as Linux [11], FreeBSD and NetBSD can be made to operate in a real-time mode, with some limitations. The trick is to take out the kernel of every non-essential feature and cut down the number of system processes dramatically. Once this is done it is almost possible to create a system with predictable real-time response.

Microsoft Windows [13], Windows/95 are relatives of DOS. They do not improve much on the features, which were required in real-time embedded computer systems. Windows/NT [10] is a good operation system for non-critical applications such as word processors and spreadsheets. In Window/NT, much of its internals is hidden, so access to low-level hardware is not possible without device drivers. It is also not possible to
configure NT so that it does not suffer from severe response latency problems. IBM's OS/2 product is based on the same core as Microsoft's Windows/NT, and it is better only in that it does allow direct access to low-level hardware. However, it easily suffers from sluggish multitasking and lacks of real-time response.

A well-written RTOS (real-time operating system) will typically have several mechanisms for error handling. In addition, most of them also have the provision to install a so-called "watchdog timer" to monitor system health. In performance critical applications, this type of periodic system checking is essential to ensure that the system does not crash or leave a mechanism somewhere in a non-controlled state.

 Most RTOS packages support a multi-threaded form of programming, which is similar to multitasking except that each thread shares global memory and environment space with all other threads. In this sense there is no memory protection, but it is not really necessary in many cases. Some RTOS do allow for the creation of isolated processes, just as with the UNIX operating systems.

An RTOS will typically allow the programmer to access all available memory and I/O space, based on the limitations of the target CPU. By definition, a true RTOS is deterministic. That is, one can predict with reasonable accuracy when a particular process event will occur by using thread priority levels, internal flags (semaphores), and thread suspension mechanisms. Interrupts can cause a small amount of delay in regards to software timing, but this is common to all RTOS. Ideally, the system latency in a real-time embedded system can only be eliminated by loading very time-critical operations onto dedicated hardware, which is a task that is almost impossible.
Compared with above operation systems, OS-9 [7], VxWorks [19], QNX, and other dedicated real-time operating systems are definitely candidates for critical real-time applications.

1.4 Contributions and Organization

This thesis presents a real-time embedded computer system based on Panther, which is a product of the company. Panther is a single-processor electronic transaction terminal device. It possesses the screen constructed by touch pad and LCD display. Customers can touch the corresponding named buttons on the screen to go through the desired processes in order. Also they can sign for receipts, contracts or identification while seeing their signatures on the screen in real time, which, in the paperless environment, mirrors the experience of signing on paper. Panther has MSR (Magnetic Stripe Reader) to accept credit card payment. The system also enables the functions of advertising and customer survey. Panther uses serial communication ports to communicate with other systems to receive instructions and send transaction data.

Firmware is the main concern here. Panther firmware system design and implementation are presented in this thesis in detail. Firmware is the software for real-time embedded computer system. Comparing with the software for desktop computers and high level applications, it is different and is more difficult to construct. It has the problems of desktop applications plus more, as illustrated by Douglass in [10]. Typically, firmware has to control the hardware devices; guarantee the responses, which are tightly bounded in time, to the external and unpredictable events in predictable manners; provide concurrency management for the executions of different task levels. Although Panther firmware is
designed for this specific system, the design process illustrates the generic method and process of analyzing and designing firmware systems for real-time embedded computer systems.

Panther firmware design also provides a practical example of using Unified Modeling Language (UML), which is the third generation of modeling language in firmware design. Comparing with the traditional approach of flow chart or state machines, modeling approach makes the system easy to be modified, maintained, and adopted to new features.

This thesis is organized in six chapters. Chapter 2 presents the relevant information on Panther system design. Chapter 3 introduces Panther firmware design, while Chapter 4 introduces firmware implementation and analyze result. Chapter 5 concludes the thesis by discussing the advantages and disadvantages of Panther system and possible ways to improve the system. In Appendices A and B, the category of relative function and the diagrams of object relationship are given. Finally, Appendix C illustrates the sample code, which is written in C++. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 2

PATHER SYSTEM OVERVIEW

The idea of Panther product, a terminal device, targeting on paperless environment and serving in retail industry was proposed from the marketing department. The engineering group evaluated the proposal and presented a general approach. The approach is to design a single processor embedded computer system equipped with I/O devices of MSR, LCD Display, Touch-Pad and SCC (serial communication controller). MSR increases the usage of the device by taking credit card payment. LCD Display provides the means for signature capture in paperless environment. Touch-Pad enables the device communicating with custom through pin-pad entries and showing the signature image. SCC provides the communication mean with outside world. With its RS232 and RS485 communication abilities, the device can talk with any current popular sale systems such as IBM POS (point of sale) system. The Vadem VG230 was chosen as the microprocessor in the system. Its high integration, as showed in Figure 2.1, simplifies the design and implementation. ISA bus is used as bus interface between the main board and the I/O devices because of its simple and low cost implementation.

The system introduced here is one of the end products, which was designed by system engineers, and implemented mainly by hardware engineering group. The system
architecture, the memory map and I/O map, and interrupt assignments are presented in their own sections in this chapter.

2.1 System Architecture

Panther has a one-chip PC platform, which is the Vadem VG230. The VG230 contains the 8086-compatible 16-bit 16MHz NEC V30HL processor. It also embodies a standard XT architecture. Panther system incorporates a build-in Magnetic Stripe Reader (MSR), a LCD Display, a back-light inverter, a serial communication controller (SCC), an IRDA transceiver, and a touch pad. Panther has 512K RAM and 512K ROM. Features of Panther terminal include:

(1) One-chip PC platform

The Vadem VG230 is adopted as the microprocessor. It contains the 8086-compatible 16-bit 16MHz NEC V30HL processor. The VG230 permits implementation of a fully compatible PC-XT. It is packaged in a single 160-pin CMOS chip and handles all PC functions including 16-bit CPU, XT core logic, CGA LCD controller, dual PCMCIA 2.1, PC Card slot, 8250 compatible UART, programmable interrupt controller 8259, internal timer 8254, DMA controller 8237, and a real-time clock (RTC). Figure 2.1 shows Vadem VG230 block diagram.
Figure 2.1 Vadem VG230 Block Diagram

(2) RAM

The 512KB battery-backed SRAM is used to maintain data, such as data of system configuration, script application, security key, custom logo and signatures. It is divided into two memory banks, 256k x 8 each. One memory bank holds odd memory address data, and the other holds even memory address data.
Data stored in RAM is supposed to be erased when Panther unit is illegally opened for security reason.

(3) ROM

512KB ROM is divided into two memory banks, 256k x 8 each. One memory bank holds odd memory address data, and the other holds even memory address data. The firmware code is stored in ROM. Panther system boot up from ROM.

(4) DISPLAY

ORION OEM3224-H3030 LCD is adopted to realize signature display, logo, and other commercial display such as advertise and customer survey screen. It supports a resolution of 320 by 240. It enables real-time display of customer signature and display of advertisement and other images.

Combining with Touch-pad, it provides the channel for custom input for function choice and custom survey. Further, combining with Touch-pad it enables the possibility of independent custom service environment with the screen driven script.

(5) MSR (magnetic strip reader)

MAGTEK MT-21 was used to support the credit card service. It supports two or three magnetic data tracks.

(6) Touch Pad

Currently Panther uses NIS/RC 872 as Touch Pad to support touch screen input and paperless signature capture. NIS/RC 872 supports a resolution of 4096 by 4096. Also, combining with LCD, it enables the custom driven environment.

(7) ISA BUS
ISA is adopted in panther system with 20-bit address bus and 8-bit data bus. ISA bus can be used as system bus and also directly as local bus. It provides the convenience for system extension to adopt new feature such as smart card.

(8) SCC (serial communication controller)

AMD AM85C30 serial communication controller supports two serial ports, one for RS232 and one for RS485 with SDLC (serial data link communication) protocol. With RS232 and RS485 communication abilities, Panther can connect with any current popular sale systems such as IBM POS (point of sale) system, and also can be driven by simply PC-based computer.

(9) IRDA (currently not implemented)

To support remote printer, Panther adopts EMIC TOIM3000 and TFDS3000, which support IrDA-compatible data transmission. They receive data from VG230 8250 UART.

2.2 Memory Map And I/O map

Panther contains 512KB RAM and 512KB ROM. Addressing is paged, via mapping registers. The memory map is shown in Figure 2.2.

2.3 Panther Interrupt

The Interrupt Controller is PC/XT compatible and based on the 8259 PIC. The VG230 provides two external interrupt request inputs, IRQA and IRQB. The remaining interrupt lines are inputs internally assigned as follows:
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFF:0H</td>
<td>BIOS</td>
<td>512KB ROM</td>
</tr>
<tr>
<td>F000:0H</td>
<td>FIRMWARE APPLICATION ROM</td>
<td></td>
</tr>
<tr>
<td>EFFF:0H</td>
<td>FIRMWARE APPLICATION ROM</td>
<td></td>
</tr>
<tr>
<td>C000:0H</td>
<td>SHADOW OF CGA BUFFER (32KB)</td>
<td></td>
</tr>
<tr>
<td>BFFF:0H</td>
<td>FIRMWARE APPLICATION ROM</td>
<td></td>
</tr>
<tr>
<td>B800:0H</td>
<td>SHADOW OF CGA BUFFER (32KB)</td>
<td></td>
</tr>
<tr>
<td>B7FF:0H</td>
<td>FIRMWARE APPLICATION ROM</td>
<td></td>
</tr>
<tr>
<td>8000:0H</td>
<td>FIRMWARE APPLICATION ROM</td>
<td></td>
</tr>
<tr>
<td>7FFF:0H</td>
<td>DATABANK RAM</td>
<td>512KB SRAM</td>
</tr>
<tr>
<td>4000:0H</td>
<td>DATABANK RAM</td>
<td></td>
</tr>
<tr>
<td>3FFF:0H</td>
<td>ACTUAL CGA BUFFER</td>
<td></td>
</tr>
<tr>
<td>3800:0H</td>
<td>ACTUAL CGA BUFFER</td>
<td></td>
</tr>
<tr>
<td>37FF:0H</td>
<td>DATABANK RAM</td>
<td></td>
</tr>
<tr>
<td>2800:0H</td>
<td>DATABANK RAM</td>
<td></td>
</tr>
<tr>
<td>27FF:0H</td>
<td>C++ HEAP</td>
<td></td>
</tr>
<tr>
<td>0600:0H</td>
<td>APPLICATION DATA, BSS, HEAP, STACK, FARDATA, AND FARHEP</td>
<td></td>
</tr>
<tr>
<td>05FF:0H</td>
<td>APPLICATION DATA, BSS, HEAP, STACK, FARDATA, AND FARHEP</td>
<td></td>
</tr>
<tr>
<td>0040:0H</td>
<td>APPLICATION DATA, BSS, HEAP, STACK, FARDATA, AND FARHEP</td>
<td></td>
</tr>
<tr>
<td>003F:0H</td>
<td>INTERRUPT VECTOR TABLE</td>
<td></td>
</tr>
<tr>
<td>0000:0H</td>
<td>INTERRUPT VECTOR TABLE</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.2 Panther Memory Map

- **IRQ0** System Timer Interrupt
- **IRQ1** Keyboard (not used)
- **IRQ2** RTC Alarm
- **IRQ3** SIO programmed to appear at 2F8H-2FFh
- **IRQ4** SIO programmed to appear at 3F8H-3FFh

For IRQ7, IRQA and IRQB, they are shown as followed:
- IRQA AM85C30 SCC
- IRQB Touch Pad and CASE OPEN
- IRQ7 MSR (previous reserved for Parallel Port)

<table>
<thead>
<tr>
<th>Register/Peripheral</th>
<th>I/O Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>8237 DMA Controller</td>
<td>000H-00FH</td>
</tr>
<tr>
<td>8259 Interrupt Controller</td>
<td>020H-021H</td>
</tr>
<tr>
<td>VG230 Single-Chip PC Platform Index Register</td>
<td>026H</td>
</tr>
<tr>
<td>VG230 Single-Chip PC Platform Data Register</td>
<td>027H</td>
</tr>
<tr>
<td>8254 Timer</td>
<td>040H-043H</td>
</tr>
<tr>
<td>VG230 Serial Port (8250 UART)</td>
<td>2F8H-2FFH</td>
</tr>
<tr>
<td></td>
<td>3F8H-3FFH</td>
</tr>
<tr>
<td>Parallel Printer Port</td>
<td>I/O R/W</td>
</tr>
<tr>
<td>LPT1</td>
<td>access</td>
</tr>
<tr>
<td>LPT2</td>
<td>378H-37FH</td>
</tr>
<tr>
<td>LPT3</td>
<td>278H-27FH</td>
</tr>
<tr>
<td></td>
<td>3BCH-3BEH</td>
</tr>
<tr>
<td>CGA LCD Controller</td>
<td>3D0H-3DFH</td>
</tr>
<tr>
<td>LED</td>
<td>380H</td>
</tr>
<tr>
<td>Am85C30 Serial Communication Controller (SCC)</td>
<td>280H-286H</td>
</tr>
<tr>
<td>MEGTAK Magnetic Stripe Reader (MSR): Track 1</td>
<td>208H</td>
</tr>
<tr>
<td>Track 2</td>
<td>300H</td>
</tr>
<tr>
<td>Track 3</td>
<td>308H</td>
</tr>
<tr>
<td>Touch Pad</td>
<td>200H-207H</td>
</tr>
</tbody>
</table>

Figure 2.3 Panther I/O Map

Panther interrupt configuration is shown in Figure 2.4.
Figure 2.4 Panther Interrupt Configuration
CHAPTER 3

FIRMWARE DESIGN

This chapter illustrates the software components by analyzing the system and looking into basic concerns for the system to fulfill the functions. The software components are packages that contain tasks. In the further, the firmware architecture design will define the tasks in each package, and interconnections in and between each package. Following firmware design is the system implementation, which is introduced in the next chapter.

3.1 System Analysis

The first concern of system analysis is communication. As mentioned in Chapter 2, SCC is adopted to provide RS232 and RS485 (with SDLC protocol) communication with serial ports. Panther, as a service device, talks with host computer through the serial ports. In order to communicate in a predictable manner, a special Panther command set is defined.

Actually, OS-9 provides a rich set of commands. There are two reasons why Panther does not use those commands. One reason is that Panther commands are supposed to be compatible with old commands for previous Penware products. The other reason is that OS-9 commands do not cover all the need for Panther. Adding new commands to OS-9 command set needs modification of OS-9 system, including its shell and system
management. This would be a heavy burden for the developers. In Section 3.4, Panther command definition and format will be introduced in detail.

A communication protocol is necessary to have a reliable communication. It is also the lesson learned from the old product. In the old product, naked data was going forward and backward between host computer and the device. Any mistake by interfering in the communication causes misunderstanding. Time-out or, in the worst case, system reset is the only ways to recover the communication which would sacrifice the system performance. Panther communication protocol is explained in Section 3.4.

To avoid error in communication and illegal commands in applications, communication Processor is in the role to check the receiving package and syntax of command(s) in the package.

Since Panther commands are beyond the OS-9 commands, the OS-9 shell can not be used to interpret the Panther commands. Thus, Panther self-use command Interpreter is proposed.

The functions provided by the OS-9 can not cover the needs of Panther. Since Panther is specifically designed for its own proposes, and it needs its own function base. PAL (Panther API Layer), thus, is proposed to define all the independent and basic functions.

Another concern is Command Applications. To fulfill the functionality defined by each command, more than one function defined in PAL may be needed. Command Applications is an application groups, corresponding to each command, and use the functions provided by PAL to realize the command functionality.

Firmware, as embedded software, directly deals with hardware I/O devices by device drivers. Device drivers are on the lowest level of firmware and talk with I/O devices by
sending and receiving data. Device driver is hardware dependent. Panther is equipped with five I/O devices: MSR, SCC, LCD Display, Touch Pad and Remote Printer. Panther firmware uses six device drivers to serve those devices. The functions realized in PAL access I/O devices by device drivers.

![Figure 3.1 Firmware blocks diagram](image)

Based on above analysis, Panther firmware block diagram is drawn in Figure 3.1. Each block shows the packages, which are the software components, each representing a single area of concern in the system.
Panther firmware is built on OS-9 OS. It is an application of OS-9 OS, which is briefly introduced in the next section.

The UML is used in the design of Panther firmware. The layer pattern was fully adopted in Panther firmware architecture design, as well as Interpreter design. Most of the diagrams shown in features were drawn with UML methodology and in its symbolic representations. The UML is simply introduced in Section 3.3.

3.2 Using OS-9 as the Operating System

OS-9, the product of Microwave, is a typical RTOS, which provides the mechanisms for multitasking and error handling. Its system latency is low. Also, OS-9 has some distinct characters.

(1) Modular Architecture
Memory modules are the foundation of OS-9 real-time operating system. OS-9 has been designed so that each module provides specific functions. This architecture allows individual module to be included or deleted in the operating system when OS-9 is configured for a specific application. This modularity makes OS-9 extremely scalable to fit the needs and easy to be maintained. In addition, OS-9 also allows for upgrades of the modules in the field.

(2) System Security
OS-9 operates in two distinct environments in which code can be executed. User state is the normal program environment in which processes are executed. System state offers all of the services found in user state but provides an environment in which system service
routines are executed. OS-9 provides an efficient environment for programmers to concentrate on application development while preserving user control of task switching and system configuration. This operating system provides a versatile system that alleviates the need for users to write redundant system management routines for memory, process, I/O, and file management. Processes run independently of others. If one process happens to fail, the system and other processes continue unaffected.

When one application fails it does not affect the others. This makes it easier to isolate bugs the development.

(3) User-Installable System Calls

OS-9 allows the flexibility to add features to the operating system. This can be extremely important if the features of an in-house operating system are needed to integrate with the OS-9.

(4) I/O Support

OS-9 features a modular, unified, hardware-independent I/O system that can be easily expanded or customized. This extensive I/O supports the power to support a broad range of applications.

(5) The Multimedia Application User Interface (MAUI) Graphics

MAUI is a set of full-featured graphics APIs designed for embedded systems. It provides a robust graphical environment for applications in a small memory footprint. MAUI supports numerous I/O devices including grayscale and color LCD panels, SVGA/VGA graphics terminals, touch screens, mice, keypad/keyboards and other devices commonly found in man-machine interfaces. MAUI provides the power of high-performance graphics with low CPU overhead and fast development time.
3.3 Introducing UML

Modeling approach, as a dominating approach in large software system design to handle the complexity, has been used over three decades. The main approaches to high-level modeling have been Structured Analysis (SA) and Object Orientation (OO).

SA started in the late 1970s. It is based on classical procedural programming concept and describes system structure by function decomposition and flow of information, depicted by hierarchical data flow diagrams.

OO modeling stated in the late 1980s, and is based on object-oriented programming concepts. The system structure is described by classes and instances, relationships and roles, operations and events, and aggregation and inheritance.

There are several primary advantages of object-oriented development.

(1) Consistency of views

In object-oriented systems, the same set of modeling view is used in all phase of development. Objects and classes identified in the analysis model have direct representations in the code. So it is almost trivial to show the relationships between the definition of the problem (analysis) and its solutions (the code).

(2) Improved problem domain abstraction

Object-oriented modeling maintains the strong cohesion among data items and the operations that manipulate them, in the way that things, being described, exist in the real world.

(3) Improved stability in the presence of changes
Because object-oriented system abstractions are based on the real world, they tend to be stable. Changing requirements is usually a matter of adding or removing aspects of the model rather than a total restructuring of the system.

(4) Improved model facilities for reuse.

Object-oriented modeling includes two means for improving reuse-generalization and refinement. Generalization supports reuse by adding and extending existing components with no changes to their source code. This is the powerful notion of "programming by difference" and allows the developer to code only the things that are different.

Refinement allows the incomplete specification of objects, which are then refined by adding the missing pieces. By using different missing parts, the same basic structure is reused.

(5) Improved scalability and better support for reliability and safety concerns

Abstraction and encapsulation of object-oriented development maintains loose coupling among components, decreasing pathological coupling. Also, because of abstraction and encapsulation, the interaction of different object-oriented components can be limited to a few well-defined interfaces. This improves reliability because it is possible to control how the components interact. Additionally, it is possible to enforce more clearly and cleanly pre- and post-conditions required for making the system run properly.

Unified Modeling Language (UML) is the result of co-working of OO methodologists, in hope of bringing together the best of various OO modeling approaches. UML is a language for expressing the constructs and relationships of complex systems. It is accomplished by visual formalisms that come complete with syntax to determine what is allowed and semantics to determine what the allowed things mean. It rigorously defines
the semantics of the object metamodel and provides a notion for capturing and communicating object structure and behavior. The main emphasis is placed on topological relationships between diagrammatic entities.

UML provides methodology for modeling design for real-time embedded system, and is used in the design of Panther's firmware. UML has been adopted by many leading companies in industry, such as HTP, 3COM and others, as standard design method.

3.4 Command Definition

As Panther is designed as a server to provide custom service, it needs to understand the client in host computer and do what it is ordered to do. Thus, command set is designed which defines rich functions to be used by client in host computer to drive Panther unit. Panther understands the commands in a predicted way as defined.

There are 162 commands, so far, defined in 5 categories. System commands are used for requiring system information such as “is system available?” and “what is the firmware version?” and so on. Communication commands require communication configuration information, or sets the communication configuration to different modes. Display commands are used for graphics.

Each command consists of a 16-bit operation code followed by 16-bit argument length, followed by optional argument data as shown below:

<table>
<thead>
<tr>
<th>Operation code</th>
<th>argument length</th>
<th>optional: argument data field</th>
</tr>
</thead>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The argument length specifies the total length of the arguments in argument data field.

If the command does not have any argument data, the argument length value is set to zero, and the argument field is omitted.

The argument field, if it exists, may consist of one or more arguments. Each argument is either fixed-length or variable-length. The size of each fixed-length argument is determined by command type. The size of each variable-length argument is specified by the 16-bit argument length, which is followed by the actual argument data.

The following diagram shows the format of a command with three arguments: the first is a variable-length argument (arg1) followed by two fixed-length argument (arg2 and arg3):

<table>
<thead>
<tr>
<th>Operation code</th>
<th>arguments length</th>
<th>argument data field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>argument data field</th>
</tr>
</thead>
<tbody>
<tr>
<td>len1</td>
</tr>
<tr>
<td>arg1</td>
</tr>
<tr>
<td>arg2</td>
</tr>
<tr>
<td>arg3</td>
</tr>
</tbody>
</table>

Among 162 commands, 83 commands are supposed to have returning data by Panther after being executed to inform the host computer of the results. The format of the returning data consists of a stream of result arguments (there are no operation code and argument length fields). Each result argument is either fixed-length or variable-length. The size of each fixed-length argument is determined by its type. A 16-bit argument length sub-field followed by the actual argument data specifies the size of each variable-length argument.
When each command was sent to Panther from host computer, the 16-bit value such as operation code and argument length is sent by order of low byte then high byte.

Following is an example of using display command to display the text message "HELLO" at the horizontal column of 8 and the vertical row 16 on the LCD display. The command string should send like:

\$80\$32\$0B\$00\$08\$00\$10\$00\$05\$00HELLO

To simplify this example lets view the bytes as follows:

<table>
<thead>
<tr>
<th>Operation-code</th>
<th>arguments length</th>
<th>argument data field</th>
</tr>
</thead>
<tbody>
<tr>
<td>$80$32</td>
<td>$0B$00</td>
<td>$08$00$10$00$05$00HELLO</td>
</tr>
</tbody>
</table>

The argument length \$000B (11 in decimal) represents the full length of all of the arguments in bytes.

In the argument data field, as shown in the following table, the first two bytes represent the horizontal column position starting with the low byte followed by the high byte. The third and fourth bytes represent the vertical row position, and as always the low byte comes first followed by the high byte. The next two bytes are the length of the variable data field that is followed, which in this case is the character string "HELLO". The next five bytes consist of the bytes that represent the string of characters we want to display.

<table>
<thead>
<tr>
<th>argument data field</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINT data type</td>
</tr>
<tr>
<td>column</td>
</tr>
<tr>
<td>$08$00</td>
</tr>
</tbody>
</table>
3.5 Communication protocol definition

3.5.1 Packed Data Format

Data I/O to Panther is in the packed format. The packed format makes communication robust. The data is packed into frames, which consist of a header and an optional message.

[HEADER\ MESSAGE]

The Host sends data to the pad as frames. These frames can contain one or more commands embedded in the Message. The Pad will respond to all host frames with an ACK or NAC frame. This is used to tell the host the receive status of the previous frame. An ACK frame is sent when the host frame is received and the CRC is correct. A NAC is sent when the CRC is not correct, or all data is not sent in a specified time. The host should re-send the frame if a NAC is received. The sending of Host Frames has to be synchronous with the pad ACK/NAC. A new host frame should not be sent before the host receives an ACK or NAC.

Along with a communication acknowledgment, each command from the host will generate a response from the pad. This response will either be the data requested by the host or a command completion status. This return result will be asynchronous. Whenever data is ready or the requested function has been run (i.e. load logo) the result will be returned (note that the response to all commands is only in the packed mode).

3.5.2 Header Description

The header is the same for host and pad communication. The FRAME byte will identify the frame type.
### [FRAME | ADDR ]

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FRAME</td>
<td>see *1</td>
<td>Frame type</td>
</tr>
<tr>
<td>1</td>
<td>ADDR</td>
<td>0-255</td>
<td>Device address of Panther target (0 default, set at configuration)</td>
</tr>
</tbody>
</table>

*1 Frame Types

- 0x01 Host Command Frame
- 0x02 ACK frame
- 0x03 NAC frame
- 0x04 Response frame

### 3.5.3 Host Message Description

The Host command frame will be immediately followed with the following type of message. This data field in this message contains one or more Panther commands.

### [Length | Data | CRC]

<table>
<thead>
<tr>
<th>Byte #</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>LENGTH</td>
<td></td>
<td>Length of data</td>
</tr>
<tr>
<td>2-n</td>
<td>DATA</td>
<td>See *2</td>
<td>CMD data</td>
</tr>
<tr>
<td>n+1</td>
<td>CRC</td>
<td></td>
<td>CRC of Message</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
*2 Host CMD Data Description

Multiple commands can be embedded in the data. Each command group is delimited by a STX and an ETX.

<table>
<thead>
<tr>
<th>STX</th>
<th>TAG</th>
<th>Priority</th>
<th>CMD</th>
<th>ETX</th>
</tr>
</thead>
</table>

Byte # Name   | Value   | Description
---|---------|-----------------
0  | STX     | 0x2F         | Start of text
1  | TAG     | Mod 256      | Cmd tag used to sync command with response
2  | Priority| 0-3          | Run priority of command 0-low 3- high
3-n| CMD     |              | CMD
n+1| ETX     | 0x2E         | End of text

3.5.4 Pad Return Message Description

The response frame will immediately be followed by the following type of messages. This data field in this message contains the response.

[TAG | Length | Data | CRC]

Byte # Name   | Value   | Description
---|---------|-----------------
0  | TAG     | Mod 256      | Command tag used to sync command with response
1-2| LENGTH  |              | Length of data
3-n| DATA    | See *2       | Response data
n+1| CRC     |              | CRC of Message

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
3.6 Firmware Architecture

In Section 3.1, the general areas of concerns were illustrated by packages. Those packages in turn contain other packages and tasks. Panther firmware architecture design is the process of designing those packages, tasks, and their interconnections. The primary diagrammatic elements are shown in Figure 3.2.

![Diagram](image)

Figure 3.2 Primary Diagrammatic Elements

3.6.1 Communication Processor

Communication Processor, as shown in Figure 3.3, contains two sub-packages. One is for Communication, and another is for Checking. There are two tasks assigned to communication: Receive and Send. Receive is designed to receive a message from outside of Panther through the serial port. It puts the receiving data in receiving message buffer. Send is designed to send data, from sending message buffer, back to the host computer system through serial port.
Checking also contains two tasks. Protocol task checks the header and CRC of the receiving message packages. Illegal header or wrong CRC leads to abandoning of the receiving package and negative acknowledgement back to the host.

![Communication Processor Package Diagram](image)

Figure 3.3 Communication Processor Package Diagram

3.6.2 Panther Interpreter

Panther Interpreter is designed to process the commands received from the host. It has three tasks to do, which are assigned to two sub-packages. Task Queuing manages the command dispatching queue. It puts each coming command into the queue in proper position according to its priority.

Dispatching task is supposed to dispatch each available command in the dispatching queue immediately by creating the thread with the corresponding Command Application
function. When two available commands in dispatching queue require the same hardware resource, the instant dispatching becomes unreasonable and leads to system misbehavior. Resource task is designed to solve the reusing of hardware. When a command, which is engaged on a hardware target, such as MSR or Touch-pad, is dispatched, the following commands in the queue, which are functionally relative to the occupied hardware, are held in the queue until the hardware is released. The diagram of Panther Interpreter Package is shown in Figure 3.4.

3.6.3 Command Applications

In Command Applications package, there are 162 tasks. Each of them corresponds to each command defined by Panther. The Command Applications is the UI from Panther Interpreter point of view, and is the application package from PAL point of view. Figure 3.5 shows Command Applications Package. Each block in the package represents each application, which is based on PAL to fulfill the functionality defined by corresponding command.

![Panther Interpreter Package](image.png)

Figure 3.4 Panther Interpreter Package

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
3.6.4 PAL

PAL defines thirty-two independently basic functions, which distributes in the tasks of eleven sub-packages. PAL package is illustrated in Figure 3.6.
The Touch-Pad package provides signature functions to handle ink data capture from the panther's touch pad.
The functions are built on a set of objects, which handle various elements of signature capturing. The main Object is the PAD object, which handles the attributes and actions of the touch pad. The PAD\_XXX() PAL functions are used to access the attributes and actions of this object.

When the pad is on, Ink data can be captured, saved, displayed and output to the host. These functions are accessed by the INK\_XXX() PAL functions. Objects such as an Ink object, Pt Manager object, internally control how the Ink is manipulated. The Touch-Pad function state model is shown in Figure 3.7.

The MSR package supports functions to turn on, read, or turn off MSR. When the unit is first powered on, the MSR will be in the off state with no data available. A call to the Msr\_On() function will then turn on the MSR driver and enable the MSR Interrupt service routine. While the MSR is on, data will be captured when a card is scanned. The data is then checked to determine if it is valid or not. If the data is valid the MSR will go to the Off state with data available, this data can be accessed with the Msr\_Get() function. If the data is invalid, the MSR goes back to the on state and waits for another card swipe. An audio response will be given for both a good and bad read. Data validity will only be checked for Tracks of interest, and this can be specified as tracks to read using Msr\_Set().

Prior to turning on the MSR the Msr\_Set() function can be called to change the characteristics of the MSR reader. The number of tracks to read and the format of the track data returned can be set with this function. The MSR settings can not be set when the MSR is on. Figure 3.8 shows the MSR function status diagram.

The default configuration is all tracks read and returned in VISA ASCII format.
Figure 3.7 Touch-Pad Function State Model

The functions provided in PIN package are used to get a user PIN in a Debit transaction. The PIN is encrypted as it is entered to form an encrypted PIN block. There are two methods of encryption supported, Master/Session Key and DUKPT. Also the PIN Block can be in either VISA standard ASCII format or binary.
The PIN_SET() function sets the variables associated with the PIN Pad session. The encryption method, PIN block format, Screen Title and Session Time out are all set with this function. This function has to be called, as there are no default value. This function will create the PIN-Pad object. Only one PIN-pad object can exist at a time. If the key for the method selected is not loaded, a PIN-Pad object will not be created and a Pin session can not be started.
The PIN_ON() function starts the PIN entry session. It displays the PIN Pad prompt and waits for the user to enter the PIN and select a button (Enter, Clear, Undo or Cancel). The “*” symbol will be entered in the display to represent a PIN character.

The PIN_OFF() function will end the PIN entry session without a PIN being encrypted and saved. The functions Pin_DKeyLoad, Pin_MKeyLoad, Pin_SMKeyLoad, Pin_SKeyLoad are used to load the Keys for the PIN encryption.

The Audio Function defined in Audio package will provide predefined bell tones which are accessible with the Audio_Bell() function and also any tone can be created by setting the frequency and duration with the Audio_Tone() function.

Panther treats the ISA bus as a generic I/O device, buffering data sent and received from this bus. ISA package provides read and write functions to allow Panther to communicate with the attached device.

Specific to the Smart Card Reader/Writer the ISA_Write function will place the bytes to be written to the smart card in the ISA out_buffer and then send them out. The format of these commands will be specific to that device. The ISA_Read() function will read a number of bytes from the ISA in buffer which will be filled with the response from the smart card reader. The ISA_Status() can be used to determine which smart card reader is attached to the panther.

Time package provides several functions to set up a digital time of day clock with year, month, day, hour, minute, and second. 1/10 sec will be provided. The clock will stop when the device is powered down. On power up the default time will be Jan 1 year 2000 12:00:00 AM. The Time_Set() is used to set all values but the 1/10 sec and Time_Get() will
return all time fields. The Time can be displayed any where on the screen as just the time, just the date, or both time and date. Seconds will not be displayed.

Communication Package concerns communication issues. Communication can be done through RS232 or RS485 interface. It is configured automatically through hardware. However, the firmware needs to know the configuration in order to handshake properly as there are significantly different in the handshaking.

For RS232 communication, it is possible to configure the port to different parameters. These parameters include the following:

Number of data bits - 7 or 8
Number of stop bits - 1 or 2
Parity - None, odd or even

For RS485 communication, the port cannot be configured and has the following fixed parameters:

Baud rate - 9600
Number of data bits - 8
Number of stop bits - 1
Parity - None

All the handshaking signals will be supported by firmware.

In the RS485 communication, handshaking is done byte by byte for transmit data. When there is data to be transmitted to the host, a request-to-send (RTS) signal needs to be activated first and wait for the clear-to-send (CTS) signal to be active before the data byte can be sent out.
Display package supports display functions. When the unit is first powered on the display will be in the off state. A call to the Disp_On() function will then turn on LCD, the display driver, and initialize MAUI. After the Display is on, all the display functions could be called.

The Display is a shared resource that should only be used by one process at a time. A semaphore is used to synchronize the display.

Display Bitmap function supports the Windows BMP format. OS-9 MAUI supports IFF image format, which is used less often now, so the Windows BMP format is selected. If the size of the image is larger than the screen size, the larger part is truncated.

Display Text function supports fonts of 6x8, 8x8, 8x12, and others. MAUI supports UCM font format, which size is 14x20. The file management function is assigned to File package. The data, such as Bitmaps, can be saved to a binary file. The data in these files can then be displayed or output to a communication port. File once created will always exist until deleting function is used.

3.6.5 Interconnection

Panther firmware packages are designed in layered pattern, as shown in Figure 3.9. In Panther firmware architecture pattern, there are client-server relationships among the package layer. The package layers are actually ordered hierarchically from the most abstract of system problem domain, represented by raw data of message, down to the most concrete of function executions. Two benefits are the result of this architecture. One is reusability. It is possible to use the same low-level layers in different contexts because
they know nothing of their client. Another is portability. Since the lower-level layers offer a well-defined set of interfaces, they can be replaced with different lower-layer making the entire subsystem easily portable to other physical environment.

Driver layer contains a set of drivers to deal with Panther I/O devices. It provides the interface between the firmware system and the hardware system with basic initialization, read, and write functions. It is hardware dependable and for same I/O devices, it is reusable for different up level system.

PAL layer uses the Driver layer functions and OS-9 operating system functions in the definitions of the basic objects, whose behaviors are fundamental functions for Panther firmware system. It depends on the layer below. It is reusable for different up level applications.

Command Applications layer is a set of command applications. Each application in this layer realizes each system function defined by the corresponding command. Modifications on this layer for changing command definition or adding new commands does not require change on its lower layers. For the same set of system functions, it is reusable for different interpreter. The disadvantage of this layered architecture is the loss of performance. Because the execution path must pass through five layers in order to invoke the required service, comparing with calling the service directly, it is inefficient. Also because lower layers know nothing about higher layers, they are general and not able to apply any optimization that requires knowledge of the higher layers.
Figure 3.9 Panther Firmware Layered Architecture pattern
CHAPTER 4

FIRMWARE IMPLEMENTATION

In this chapter, some critical issues such as time deadline, task priority, and thread assignment are introduced. The implementation codes are not illustrated, because of their large bodies and also some confidential concern. Some pseudo codes and state machine models are given to illustrate some key functions of execution processes. Also in Appendix C, sample codes are presented with modification.

4.1 Thread Assignments

There are three threads consistently running when Panther is in service. They are Communication Receiving Thread, Communication Sending Thread, and Dispatching Thread. The Dispatching Thread dispatches the commands waiting in the dispatching queue. When a command is dispatched in its turn, the Dispatching Thread creates a sub-thread correspondingly and the command’s function(s) will be executed in this sub-thread which is generically named Command Execution Thread. A Command Execution Thread terminates when the corresponding command execution is completed or canceled.
4.1.1 Communication Thread

The tasks, except sending in Communication Processor package are assigned to Communication Receiving Thread. Figure 4.1 shows the execution process of Communication Receiving Thread.

![Communication Receiving Thread State model](image)

Figure 4.1 Communication Receiving Thread State model

4.1.2 Communication Sending Thread

Sending tasks in Communication Processor package is assigned to Communication Sending Thread. Communication Sending Thread is in charge of sending the data out to the remote host by serial port. All the Command Execution Threads and Communication
Sending tasks in Communication Processor package is assigned to Communication Sending Thread. Communication Sending Thread is in charge of sending the data out to the remote host by serial port. All the Command Execution Threads and Communication Receiving Thread use Pipes, which are the inter-thread communication method provided by OS-9, to transfer data to Communication Sending Thread in order to send communication acknowledgement, system information, and the results of command execution. Figure 4.2 shows the states in this thread.

![Communication Sending Thread State Model](image)

Figure 4.2 Communication Sending Thread State Model
4.1.3 Dispatching Thread

The tasks in Panther Interpreter package are assigned to Dispatching Thread. In this thread, the hardware resources, such as MSR and Touch-Pad, are managed. Each time when resource required command has been dispatched, the resource occupied flag is set. The next command, which uses the same hardware resource, is held but staying in the front of the dispatching queue until the flag is turned off. The flag is turned off in the Command Execution Thread when it releases the resource. Figure 4.3 illustrates the states and the execution process of Dispatching Thread. Dispatching Thread does not inspect the execution of its sub-thread, which is Command Execution Thread.

4.1.4 Command Execution Threads

Thread executes the application in Command Applications package when the relative command is dispatched for execution. Each Command Execution Thread exits when the application terminates.

4.2 Time Deadlines

Real-time systems encompass all devices with performance constraints. Hard deadlines are performance requirements that absolutely must be met. A missed deadline constitutes an erroneous computation and a system failure. In those systems, late data is bad data. Soft real-time systems are constrained only by average time constraints. In these systems, late data is still good data. According to these definitions, Panther system is a soft real-time system.
Figure 4.3 Dispatching Thread State Model

In panther system, the input data is periodical and predictable. All the devices talk with outside world in predictable manners by commands, such as MSR, Touch-pad. Now, in order to analyze the processor and I/O device input and output ability, two assumptions are made. One assumes that I/O devices are zero wait state devices. Another assumes that all threads created are eligible for the CPU at all times. Also, a slice is defined as the amount of time a thread may remain in the CPU before the Kernel decides to perform a context switch.
Panther uses VG230 microprocessor, which adopts 16MHZ CPU. It also uses ISA bus with 8-bit data bus. All transfers performed over the ISA bus are synchronized to an 8.33MHZ bus clock signal. It takes two cycles of the bus clock, if the target device is a zero wait state device, to perform a data transfer. This equates to 4.165 million transfers per second. Since data bus used in Panther is only 8 bits wide, one byte may be transferred during each transaction. Thus, the transfer rate is 4.165 Mbytes per second.

If supposing all the commands defined by Panther, total 162 commands, are dispatched as Command Executing Threads and all threads are in execution, there are total 162 threads running in the processor. If all the threads take the same priority, they share the CPU time equally. The maximum transfer rate in each thread is 0.025242 Mbytes per second. If each thread uses one of tenth of its time to perform data transfer, it can make 0.002524 Mbytes, or 2524 bytes per second.

A slice is set to 0.1 MS in Panther firmware implementation. The maximum delay between any two threads is 16.4 MS. NIS/RC 872 Touch-Pad has maximum input ability of 197 points per second. Each point is represented by four byte data. So it may have 788 bytes per second data input. The MSR and SCC work in a serial manner. They have slower input and output abilities than 2524 bytes per second.

The analyses show that Panther system has enough input and output ability to handle its I/O devices. In the case of capturing the signature signing on the Touch-Pad and showing the signature image on the LCD in real time, the bottleneck is Touch-Pad. The maximum 16.4MS delay is too small to matter in the point of human visual ability. Actually there could not be 162 thread running simultaneously, because of the hardware resource management and slow serial port input.
4.3 Scheduling Method

Panther firmware uses aging mechanism as scheduling method. Every thread that enters the line of waiting CPU time is assigned an initial age equal to the thread’s priority. The sorting of the line is thus based on each thread’s age other than its priority. By comparing with the ages of the threads already in the line, the thread inserted is put in the line behind threads with older or same age, and before the threads with younger age.

When the Kernel needs a thread to execute, it takes the thread in front of the line, which has oldest age. And meanwhile, when the oldest thread has the CPU time, all other threads in the line age by one. When a thread’s time slice expires but does not complete its mission, it is reassigned an age equal to its priority, and sorted into the line.

All the threads, including the three consistent threads and Command Execution Thread, running in the Panther processor currently are assigned the same age 130.

4.4 Dispatching Queue Sorting Method

Panther command dispatching queue is managed by the command’s priority. The coming command is put in the queue, which is implemented as a link list, behind the commands with high priorities and in front of commands with lower priorities. Late coming commands with higher priorities may be executed earlier than the early commands with lower priority.
4.5 Drivers

The drivers in Panther are I/O device dependent. They are implemented in C and Assembly code, in simple and fast style. They are written in the same routine as illustrated in the pseudo code, by meaningful names, as followed:

Driver's pseudo code

drvmain
{
    Initialization();
    Read();
    Write();
    GetStatus();
    s();
    TerSetStatuminate();
}
end of drvmain
CHAPTER 5

CONCLUSION

This chapter presents the conclusions of this thesis. Section 5.1 lists observations made and discusses the advantages and disadvantages of Panther system. In Section 5.2, some suggestions are proposed for improving the system.

5.1 Observations

Panther is the new generation in the same product category, and possesses new capabilities. It is not only able to electronically capture the signature, but also is able to show the signature image, which mirrors the pen and paper experience. The LCD Display also enables advertising and customer survey. Adopted MSR equips this single device with credit card payment transaction capability.

Panther can be driven by most existing PCs and IBM Point of Sale systems (POS). Comparing with around $10,000 IBM POS system, Panther provides low-cost choice.

Even though the cost of Panther is about four times as much as current credit card terminals, the device will pay for itself in more accurate transactions, saved paper, and reduced record-keeping expenses.
Panther’s performance is mostly as good as expected, but not ideal. The enhancement of system stability, for instance, is in schedule. The difficulty of writing customer applications is still a problem. A tool kit for custom application programming is absolutely necessary.

From the design point of view, the programming language, C++, does not provide convenience for graphic function creation. The operating system is also lacking of easily used functions for graphic, although it has the capability. The screen driven script is hard to be modified.

Panther uses Vadem VG230 as the microprocessor. VG230 has XT architecture and it is 8086-compatible microprocessor. It addresses 1M byte memory only. After storing the operating system and firmware application, only 317K bytes are left in the system. And there is no extended memory available. The left memory is no big enough for the implementation of some memory consuming applications, such as Java script. The lack of memory limits the system functional extension.

5.2 Future Direction

Panther is designed to be a terminal device. This means it has to been driven by another computer system, which is the host. This would be an absolutely bad decision if it has been made today. When Panther-like device is equipped with Touch-Pad and LCD Display, it has a panel for communication with custom visually. Thus the device has potential ability to become standing along terminal which runs script application and is driven by custom
through the screen. It may drive host computer when necessary to require the decision
that beyond its duty, new application, necessary data, or access to the remote database.

Java fully supports graphics with a rich graph function library. Writing a screen driven
script application becomes much easier.

Future Panther-like product trends to be network-based. The stand-alone network-
based Panther-like product may finally take place of complicated and expensive IBM POS
systems in the markets and stores. Also this network-based product possesses the ability
of providing Internet services, such as Internet shopping and information search. It will
not be surprising that someday in near future the products is set in airports for airline
information service, ticket reservation, or destination service reservation. It may be
equipped with telephone, and may also be wireless.

Someday I may stand in front of it in Hawaii and turn off the stove at home in San
Jose, by touching the screen.
## APPENDIX A

### PAL FUNCTION CATEGORY

#### Signature Functions
- **Pad_On()**: Starts the pad capture process.
- **Pad_Off()**: Ends the pad capture process.
- **Pad_Set()**: Sets pad scan rate and Spike filter options.
- **Pad_Status()**: Gets Info on Pad.

#### MSR Functions
- **Msr_On()**: Starts MSR process.
- **Msr_Off()**: Stops MSR process.
- **Msr_GetData()**: Gets MSR data.
- **Msr_Set()**: Set MSR for read options.
- **Msr_Status()**: Gets MSR Info.

#### PIN Functions
- **Pin_Set()**: Sets up PIN pad.
- **Pin_On()**: Starts PIN process.
- **Pin_Off()**: Ends PIN process.
- **Pin_Get()**: Gets PIN block.
- **Pin_PadStatus()**: Get PIN pad info.
- **Pin_DKeyLoad()**: Loads DUKPT Key.
- **Pin_MKeyLoad()**: Loads Master Keys.
- **Pin_SKeyLoad()**: Loads Session Keys.
- **Pin_ClearKey()**: Clears all keys.
- **Pin_KeyStatus()**: Checks if keys set.

---

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Audio Functions
Audio_Tone() Plays specified tone
Audio_Bell() Plays standard bell sounds

Smart Card Functions
SC_Read() SC_Write() SC_Status()

Time Functions
Time_Set() Sets current time
Time_Get() Gets current time
TimeDisp() Displays current time

Communication Functions
Com_Config() Configure Com
Com_Status() Get settings of Com
Com_In() Get data from Com
Com_Out() Send Data to Com
Com_PassConfig() Set up pass through
Com_PassOn() Start Pass through
Com_PassOff() End Pass through

Display Functions
Disp_On() Turns on LCD
Disp_Off() Turns off LCD and back-light
Disp_Set() Sets display options
Disp_Status() Gets status of LCD
Disp_Pt() Display Pt on LCD
Disp_MoveTo() Set current pt
Disp_LineTo() Draw line from current pt to new pt
Disp_Txt() Display Text
Disp_Frame() Display rectangular frame
Disp_Bitmap() Display Bitmap
Disp_Obj() Display memory object
Disp_ClearPt() Clear pt on LCD
Disp_Clear() Clear All or section of LCD
Disp_Scroll() Scrolls a given area of screen

User Selection Functions
Button_Set() Sets Options for Button
Button_On() Activates button
Button_Off() Deactivates button

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Button_Clear() Removes button from display and deactivates
Button_Status() Gets button status

KPad_Set() Sets type of Key pad
KPad_On() Starts Key pad process
KPad_Off() Stops Key pad process

KPad_Status() Gets Key pad info

Encryption Functions
DES_Encrypt()

File Functions
File_Open()
File_Close()
File_Write()
File_Read()

System Functions
Sys_reset() Reset system to initial state
Sys_Status() Get system status

Debug_On Record errors and function call history
Debug_Off Stop debugging
Debug_Status() Get debug status
APPENDIX B

PART OF PAL OBJECTS RELATIONSHIP

DIAGRAMS

Pad & Ink Object Relationship Diagrams
Pad Objects Relationship Diagram

Display object relationship diagrams
Disp Bitmap

Input:
filename (*.bmp),
start (x,y)
Action: Display the image on screen

Load Image File

Parse Img Info

Load img into Pixel Mem

Set Destination Display

Update the Display

Wrap Up

MAUI support IFF image format, which is not easy to find developing
tools. Panther supports Windows BMP format right now.

Parse the image info, such as width, height, size, coding method and
palette from the windows BMP file.

The image data stored in the BMP file is in the reversed order, we need to
rearrange it and load into the mem, we also need to load the palette of the
image.

If the size of the image is larger than the screen size, the larger part will
be truncated, starting display the image from left upper corner.
**Disp Text**

Input: Font Size, FgColor, Bgcolor, start (x,y)

Action: Display the text string on screen

---

MAUI supports UCM font format, which size is 14x20 and could not be changed. Panther supports PW3100 fonts (6x8, 8x8, 8x12, ...) right now. If not give the size of the font, use the default.

Set the font's FgColor the same as the required FgColor, and set the font's Bgcolor, if not specified, use the same as the Dst's Bgcolor.

If the size of the text is larger than the screen size, it will give the error message.
Disp Clear

Input: 
Action: Display the background color of the Display

Disp Set

Input: BgColor 
Action: Set the background color of the Display

Disp Pt

Input: FgColor, Position (x, y) 
Action: Draw a point on the Display

Disp ClearPt

Input: Position(x, y) 
Action: Clear a point on the Display

Load Bgcolor from the palette

Create the Block Object

Set Dst of the block

Draw the block with the BgColor

Update the display

Destroy the block object

Create Draw Object

Set the Color of the drawing

Set Dst of the drawing

Draw the point

Update the display

Destroy the Draw Object

For Clear CMD, load the BgColor of the display, and do the same as draw a point.
To draw multi lines with head of one line connecting the end of another line we need to create an instance of the object and set up the start point of the first line, the default start point is (0, 0).

If the size of the line is larger than the screen size, it will give the error message.
APPENDIX C

SAMPLE CODE

Touch-Pad function implementation code:

```c
#include "Ink.h"
#include "Display.h"
#include "DataBank.h"
#include "thinning.h"
#include "ink2cotf.h"
#include "ink2hodo.h"
#include "ink2nlc.h"

#include <memory>

// scaling helper
#define SCALE( a, num, den ) ( (int)(((long)a * num) / den) )

struct InkRecord
{
    char id[3];
    int xdpi, ydpi;
    unsigned count;
    struct { int x, y; } p[l];
};

static const char cINK[] = "INK";

// CONSTRUCTOR/DESTRUCTOR

CInk::CInk( int aXDPI, int aYDPI, unsigned aSize ) : theDPI( aXDPI, aYDPI ), theCount( 0 )
{
    #ifdef _DOS
    aSize - 1000; // SOME STRANGE BUG!!!!
    #endif
    thePoints = new CPoint[aSize];
    theSize = (thePoints ? aSize : 0);
}

CInk::~CInk()
{
    delete [] thePoints;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CInk::CInk( const void *aBuf )
: theDPI( 0, 0 ), theCount( 0 ), thePoints( 0 ), theSize( 0 )
{
    Load( aBuf );
}

GLOBALS

PUBLIC MEMBER FUNCTIONS

---

// Draw

void CInk::Draw( CDisplay &aDisplay, int x, int y, unsigned from )
{
    if( from < theCount )
    {
        int nx, ny, ox, oy, p;
        unsigned i;
        ox = SCALE( GetX( from ), TheDisplay.XDPI(), XDPI() ) + x;
        oy = SCALE( GetY( from ), TheDisplay.YDPI(), YDPI() ) + y;
        p = 0;
        for( i = from + 1; i < theCount; i++ )
        {
            nx = SCALE( GetX( i ), TheDisplay.XDPI(), XDPI() ) + x;
            ny = SCALE( GetY( i ), TheDisplay.YDPI(), YDPI() ) + y;
            if( p )
                aDisplay.Line( ox, oy, nx, ny );
            ox = nx;
            oy = ny;
            p = IsLineTo( i );
        }
    }
}

---

// Add

unsigned CInk::Add( int x, int y, int p )
{
    if( theCount < theSize )
    {
        if( !p || !theCount )
            x = -x;
        thePoints[ theCount++ ].Set( x, y );
        return theCount;
    }
    return 0;
}

---

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
// Save
// --------------------------------------------------------
BOOL CInk::Save( CDataBank& aDataBank, const char* aVarName, int aTypeCode )
{
    unsigned size = sizeof( InkRecord ) + ( sizeof( CPoint ) * theCount - 1);
    if( aDataBank.Add( aVarName, size, aTypeCode ) )
    {
        void *ptr = aDataBank.GetData( aVarName );
        return Save( ptr, size );
    }
    return FALSE;
}
unsigned CInk::Save( void *aBuf, unsigned aSize )
{
    unsigned size = sizeof( InkRecord ) + ( sizeof( CPoint ) * theCount - 1);
    if( aSize >= size )
    {
        InkRecord *ink = (InkRecord*)aBuf;
        unsigned i;
        memcpy( ink->id, cINK, sizeof( ink->id ) );
        ink->xdpi = XDPIQ;
        ink->ydpi = YDPIQ;
        ink->count = theCount;
        for( i = 0; i < ink->count; i++ )
        {
            ink->p[i].x = thePoints[i].x;
            ink->p[i].y = thePoints[i].y;
        }
        return size;
    }
    return 0;
}

// Load
// --------------------------------------------------------
BOOL CInk::Load( CDataBank& aDataBank, const char* aVarName )
{
    return Load( aDataBank.GetData( aVarName ) );
}
unsigned CInk::Load( const void *aBuf )
{
    InkRecord *ink = (InkRecord*)aBuf;
    theCount = 0;
    if( ink && !memcmp( ink->id, cINK, sizeof( ink->id ) ) )
    {
        unsigned i;
        if( theSize < ink->count )
        {
            delete [] thePoints;
        }
thePoints = new CPoint[ink->count];
if( thePoints )
    theSize = ink->count;
}
if( thePoints )
{
    theCount = ink->count;
    theDPI.x = ink->xdpi;
    theDPI.y = ink->ydpi;
    for( i = 0; i < theCount; i++ )
    {
        thePoints[i].x = ink->p[i].x;
        thePoints[i].y = ink->p[i].y;
    }
    return theCount;
}
return 0;

// ThinTo

unsigned CInk::ThinTo( unsigned aSize )
{
    theCount = thin( ((INKPOINT*)thePoints), theCount, aSize );
    return theCount;
}

// ThinBy

unsigned CInk::ThinBy( unsigned aPercent )
{
    if( aPercent < 100 )
    {
        unsigned desired = (unsigned)(((unsigned long)theCount * aPercent) / 100);
        theCount = thin( ((INKPOINT*)thePoints), theCount, desired );
    }
    return theCount;
}

// SaveAsPackets - penware100 five byte packet

unsigned CInk::SaveAsPackets( void *aBuf, unsigned aSize )
{
    // convert point data into 5 byte packets
    unsigned size = 5 * theCount;
    if( aSize >= size )
    {
        struct packet { BYTE status, xyhigh, xlow, ylow, chksum; } *p;
        unsigned x, y;
        unsigned i;
        p = (packet*)aBuf;
        for( i = 0; i < theCount; i++ )
        {
            p[i].status = thePoints[i].x;
            p[i].xyhigh = thePoints[i].y;
            p[i].xlow = thePoints[i].y;
            p[i].ylow = thePoints[i].y;
            p[i].chksum = thePoints[i].y;
        }
    }
    return theCount;
}
for( i = 0; i < theCount; i++ )
{
    // build the next packet
    if( thePoints[i].x != 0 )
    {
        p[i].status = 0x98; // pen down!
        x = (unsigned)thePoints[i].x / 4;
        y = (unsigned)thePoints[i].y / 4;
    }
    else
    {
        p[i].status = 0x88; // pen up!
        x = (unsigned)(-thePoints[i].x) / 4;
        y = (unsigned)(-thePoints[i].y) / 4;
    }
    // set x/y bytes
    p[i].xyhigh = ((x & 0x380) >> 4) | (y >> 7);
    p[i].xlow = 0x7F & x;
    p[i].ylow = 0x7F & y;
    // set checksum byte
    p[i].chksum = 0x7F & (p[i].status ^ p[i].xyhigh ^ p[i].xlow ^ p[i].ylow);
}

return size;
}
return 0;

unsigned CInk::SaveAsCOTF( void *aBuf, unsigned aSize )
{
    return ink2cotf( aBuf, aSize, (const INKPOINT*)thePoints, theCount );
}

unsigned CInk::SaveAsHodo( void *aBuf, unsigned aSize )
{
    return ink2hodo( aBuf, aSize, (const INKPOINT*)thePoints, theCount );
}

unsigned CInk::SaveAsNLC( void *aBuf, unsigned aSize )
{
    return ink2nlc( aBuf, aSize, (const INKPOINT*)thePoints, theCount );
}

unsigned CInk::SaveAsPoints( void *aBuf, unsigned aSize )
{
    unsigned size = sizeof( *thePoints ) * theCount;
    if( aSize >= size )
    {
        memcpy( aBuf, thePoints, size );
        return size;
}


```c
    return 0;
}
/*EOF*/
```


VITA

Graduate of College  
University of Nevada, Las Vegas

Chenyong Huang

Home Address:  
4214 Claymont St. #4  
Las Vegas, NV 89119

Degree:  
Bachelor of Science, Engineering, 1986  
Liaoning University of Industry, P. R. China

Thesis Title: Penware Panther: An Embedded Computer System for Real-time Applications

Thesis Examination Committee:  
Chairperson, Dr. Shahram Latifi, Ph. D.  
Committee Member, Dr. Yahia Baghzouz, Ph. D.  
Committee Member, Dr. Eugene Mcgaugh, Ph. D.  
Graduate Faculty Representative, Dr. Laxmi Gewali, Ph. D.