Implementation of the MASC Information Appliance

by

Mark Lee Huang

S.B., Massachusetts Institute of Technology (1999)

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

Master of Engineering

at the

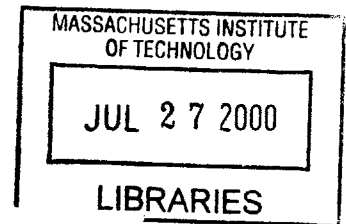
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2000

© Mark Lee Huang, MM. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

ENG



Author ....
Department of Electrical Engineering and Computer Science
January 27, 2000

Certified by ....
Srinivas Devadas
Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Implementation of the MASC Information Appliance

by

Mark Lee Huang

Submitted to the Department of Electrical Engineering and Computer Science
on January 27, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

In this thesis, I designed and implemented the run-time operating system (RTOS) for the MASC system. The Modular Appliance Super Computer (MASC) system proposes a novel solution to the problem of efficient information appliance (IA) design. To add functionality to a MASC IA, one simply inserts inexpensive, credit card-sized computer and peripheral cards into a standard backplane embedded in the IA. The RTOS software running on the computers, or hostcards, enables the hostcards to work together with the peripherals and the appliance to add intelligence to the overall device.

Thesis Supervisor: Srinivas Devadas
Title: Professor

Acknowledgments

Cheng Cheng and Sandeep Chatterjee produced the first working hostcard and backplane and paved the way for the current run-time operating system to be developed. Andrew Sutherland and Benjamin Chambers performed most of the work and coding involved with the demonstrations described in Chapter 4. Sandeep Chatterjee and Todd Mills were the major developers of the FPGA code described throughout Chapter 3. The ARM/Linux community—in particular, Nicolas Pitre and Russell King—are chiefly responsible for making embedded Linux a reality. I would like to thank all of these colleagues and friends for their help in completing this work.

Several of the figures used in this work are courtesy of the following:

- Figure 2-1: courtesy of AMP, Inc.
- Figure 1-2: courtesy of Mitsubishi Electronics America, Inc.

Contents

1 Introduction **8**

1.1 The MASC system 11

1.1.1 The MASC hostcard 11

1.1.2 The MASC backplane interface 13

1.1.3 The MASC RTOS 15

1.2 Thesis organization 16

2 The MASC prototype **17**

2.1 MASC hostcard 18

2.1.1 Specifications and features 18

2.2 MASC software 19

2.2.1 ARM/Linux 20

3 Kernel modifications and extensions **22**

3.1 Boot loader 23

3.2 Flash EEPROM driver 25

3.3 LCD driver 27

3.4 MASC driver 28

3.5 CardBus services 29

4 User space applications **32**

4.1 Filesystem 32

4.2 Graphics library 33

4.3 DAC library 34
4.3.1 Demo: MP3 player 35
4.3.2 Demo: Motor control 35
4.4 ADC library 35
4.4.1 Demo: Sensor Control 37
4.4.2 Demo: PDA 37

5 Development system and process **39**
5.1 ARM/Linux toolchain 40
5.1.1 Cross compiler 40
5.1.2 Simulator 40
5.2 Development process 41

6 Conclusion **42**

A Tables **44**

B Figures **46**

List of Figures

1-1 A MASC card. 10

1-2 A MASC card being inserted into an information appliance. 11

1-3 A prototype MASC backplane. 14

2-1 A prototype hostcard. 18

2-2 Block diagram of hostcard. 19

4-1 Car block diagram 36

4-2 Car 36

4-3 PDA block diagram 38

B-1 Filesystem storage option 1 46

B-2 Filesystem storage option 2 47

B-3 Filesystem storage option 3 47

B-4 SPI buffer system 47

List of Tables

A.1 Filesystem storage options 44

A.2 Driver dependencies 45

Chapter 1

Introduction

The hottest trend at the 1999 COMDEX Show was the "information appliance" (IA). From set-top boxes to MP3 players to networked refrigerators, any product with a microprocessor inside it and a link to the Internet was examined with keen interest. Exactly how these information appliances would fit into the household, how they would be used, or even what they should do were questions that remained unanswered beneath the hype [15].

However, even with limited information, at least three characteristics common to every information appliance could be distinguished. Virtually all of the devices expected to become part of a home network. Whether that network was based upon Bluetooth, FireWire, or wireless Ethernet[1] seemed to matter little to developers, so long as *information* appliance and *Internet* appliance remained synonymous terms.

Second, almost all of the devices used low-power embedded processors. With the occasional exception of the so-called "thin client" (generally, a stripped-down desktop computer intended for network applications), all of the information appliances sported a new breed of processors beneath their hoods. Cost was no doubt a major factor in this decision, but, as one COMDEX attendee pointed out, "Unlike PCs, the most important consideration [for information appliances] is not sheer CPU power, but MIPS per megawatt." [15]

[1] Both Bluetooth and wireless Ethernet are frequency-hopping spread-spectrum (FHSS) radio transmission schemes. FireWire, or IEEE 1394, is a fast peripheral communications protocol.

The final common aspect was a classic "agreement to disagree" over the choice of operating system and application software. High-end personal digital assistants (PDAs) tended to run Microsoft's Windows CE, yet this kind of product represents only a small fraction of the IA market. Dozens of other operating systems compete for market share in this almost chaotic environment. End users generally find this kind of chaos unappealing, yet for developers, a resounding lack of standards or even common goals is a good sign of a profitable niche in an industry practically strangled by Microsoft and Intel. Because of their unique software requirements—small code bases, efficient execution, and hard real-time reliability—information appliances filter out casual ports of legacy operating systems and demand near-perfection in new software design.

Thus, though developers seem to recognize the crucial common aspects of IAs—connectivity, low-power computation, and new, efficient software—unanswered fundamental questions still remain. To whom are IAs targeted? What should IAs do? How should they be built? Formulating answers to these questions would help to justify many of the arbitrary decisions IA developers routinely make, yet no one has approached the design of an IA with a logical plan that does so.

The Modular Appliance Super Computer (MASC) project views at least the last unanswered question as an infrastructure problem, rather than as a problem which a software hack or hardware advance could solve. The first two questions—what IAs should do and to whom they should be targeted—are important questions that are often overshadowed by implementation details. The MASC project aims to provide one solution to this problem—how information appliances should be built—to give creative developers more time to consider strongly their answers to the first two before producing hopefully useful devices.

Because the problem is more complicated than simply choosing one processor or OS over another, MASC takes a "ground-up" approach to producing information appliances rather than a "stripped-down" one. The stripped-down approach comes into play when cramming a PC into a would-be IA inevitably fails at some point in the development process. At this point, the developer incrementally replaces the CPU,

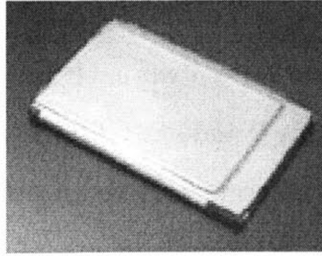


Figure 1-1: A MASC card.

the operating system, features, and functionality until the price and performance are both marginally acceptable.

In contrast, MASC's ground-up approach starts at the beginning. Instead of fundamentally changing the appliance by embedding a computer within it, the MASC system proposes isolating the functionality of the computer from that of the appliance by implementing a standard interface through which the computer and related peripherals provide their services. Adding computational power to the appliance thus becomes a matter of literally plugging it into the appliance, rather than embedding it.

MASC's vision is that of credit card-sized devices (see Figure 1-1) interacting with each other through this standard interface to provide customizable, upgradeable, and modular functionality. These devices can be intelligent parts such as computer cards (or *hostcards*) as well as more traditional peripherals such as network cards, multimedia cards, or interface adapters. Any or all of them can be used to create information appliances from regular appliances without the intermediate step of figuring out how to embed the CPU. For example, a television equipped with the standard interface—the MASC backplane—could require a hostcard to provide more powerful digital decoding capabilities. If a network card is later added, the television then becomes capable of accepting real-time digital feeds from the Internet and is transformed into an information appliance. If more processing power is needed, another hostcard can be added; if the hostcard is needed elsewhere, it can be removed (see Figure 1-2).

Using the MASC system, a typical IA can still retain connectivity through a standard network adapter, low power consumption through the inherent "functionality-

Figure 1-2: A MASC card being inserted into an information appliance.

on-demand" feature of the backplane, and flexible software options since the computer is no longer a static part of the IA.

1.1 The MASC system

With the MASC system comes costs and requirements, of course. Generalized, modular computation always requires more raw CPU power than application-specific computation. Standard interfaces must exceed the bandwidth requirements of the worst-case application. Last but most not least, computation with intentionally wide application requires software that is as reliable, extensible, and customizable as the appliance with which it is being used.

1.1.1 The MASC hostcard

In the past decade, embedded microprocessor technology has advanced at an unprecedented rate. Emerging from an era when "embedded" simply meant "battery-powered," modern embedded processors have evolved from stripped-down versions of standard RISC and CISC designs to full-fledged, highly sophisticated designs. The Intel StrongARM, for instance, one of the more popular choices for embedded applications today, now clocks in at over 600 MHz while consuming less than half a watt of power [11]. A more traditional processor such as the Intel Pentium III, on the other hand, may consume up to 30 to 35 watts of power while providing only 2 to 3 times the performance [5]. This figure does not even account for the power drawn by the external circuitry required to match the StrongARM's built-in peripherals.

What has driven this advancement has been a gradual acceptance of the need for high-end (but low-power) computation in small applications. Ten years ago, it would have been ludicrous to suggest that a telephone—even a cellular one—could ever need more than a few inductors, capacitors, and possibly some glue logic. In stark contrast to that vision, over 70% of cellular phones today contain a 32-bit ARM RISC core[2] [7]. Some would say that this kind of power is overkill for the ability to make a simple phone call, but the number of heavyweight features in portable phones is overshadowed only by the continual demand for more. Complex digital processing, including noise cancellation, compression, and encoding, is now performed regularly by even ultra-portable phones. Miniature World Wide Web (WWW) browsers are becoming more common on networked phones, and full PDA capability is available on others.

Other applications besides cellular phones benefit from high-end computation. With powerful CPUs at their disposal, developers can

- Move entire peripherals from hardware to software.
- Keep up with bandwidth advancements.
- Implement real-time processing of data (multimedia streams).
- Produce better user interfaces.
- Support emulation of legacy protocols.
- Use off-the-shelf operating systems.
- Program in high-level languages.
- Realize faster times-to-market.
- Realize longer product lifetimes.

[2]And ARM is only a single company—dozens of others, including MIPS Technologies, Texas Instruments, and Qualcomm, are considering spinning their own embedded CPU designs as well.

Information appliances could benefit from features like these, and it is fortunate that killer applications like wireless communication have brought embedded CPUs to this level. What is amazing, however, is that even with this kind of power available to them, IA developers have not attempted to free themselves from an embedded design mentality. In the past, when 200-line hand-written assembly code ran on 4-bit processors, there was indeed a strong argument to embed the processor within the appliance: the system simply wasn't worth generalizing when all 200 lines of the code were application-specific anyway. Today, IA developers implant high-end general-purpose CPUs like the StrongARM into their devices yet still employ application-specific firmware and software.

MASC proposes an end to this behavior by lifting the CPU out of the appliance entirely and into a standard package: the MASC hostcard. Removing the CPU from the development picture makes it possible to concentrate both hardware and software efforts on implementing desired functionality in an information appliance, thus answering at least one of the fundamental questions.

1.1.2 The MASC backplane interface

Removing the CPU from the appliance, however, creates a new problem of communication. A generalized system like MASC must support the same functionality that a targeted embedded system does, except with the CPU *outside of* the appliance. The interface between the two must thus feature

- High bandwidth.
- Proven reliability.
- Acceptance as a standard.
- Extensibility.

The real value of the MASC system lies in the fact that an interface with these features need not be limited to providing communication only between the CPU and

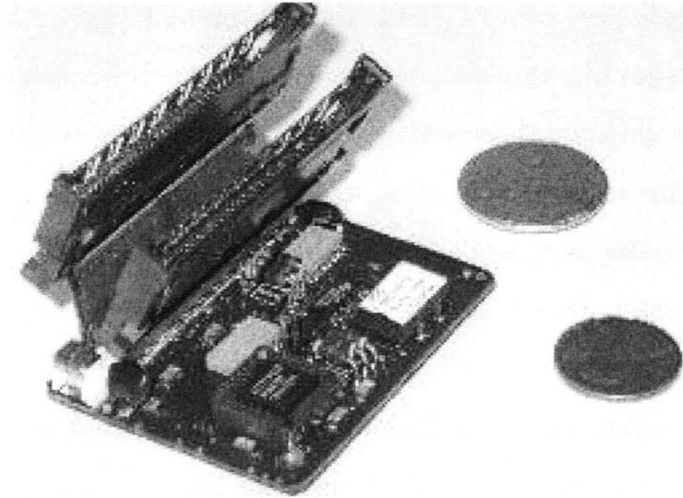


Figure 1-3: A prototype MASC backplane.

the appliance. The interface may also allow peripherals to be attached to the interface to provide additional functionality.

The MASC backplane (see Figure 1-3) serves as this standard interface. The backplane uses the 68-pin PC Card (formerly PCMCIA) format to support both 16-bit and 32-bit ("CardBus") transfers between any two devices on the bus. CardBus uses the same bus speed (33 MHz), data width (32 bits), and protocol as standard PCI[3]. Additionally, dozens of manufacturers already produce PC Card peripherals ranging from network adapters to non-volatile storage to sound cards—often, the differences between the PC Card and PCI versions of the peripherals are negligible and the same software drivers can be used.

The MASC vision is simply to add computation to the list of supported PC Card functions. This vision requires not only hardware like powerful embedded CPUs to run on MASC hostcards, but also software like a run-time operating system to control transactions between the CPU, the backplane, and peripherals on the backplane bus.

[3]While the PCI specification does not rule out 66 MHz bus speeds and 64-bit data widths, they are only rarely seen.

1.1.3 The MASC RTOS

Every information appliance developer faces the problem of choosing a run-time operating system (RTOS) to meet software goals. Thus far, no single operating system vendor has been able to claim dominance in the information appliance market. Microsoft's Windows CE has failed to take even a foothold in the war for the PDA, while other, smaller vendors such as QNX and Wind River tend to be successful only in the mission-critical markets.

This lack of dominance is not without reason. The major design consideration evident in OSes like Windows CE is that of familiarity, while for OSes like VxWorks it is extreme leanness. Windows CE looks, feels, and operates much like its bigger, tougher siblings, and this is exactly the reason why Windows CE might never become a dominant IA platform. Information appliances are inherently unfamiliar devices. They are not stripped-down PCs, nor are they limited to being handheld PDAs, and nor will they necessarily always require a user interface. For example, why run Windows CE on an MP3 player[4] when no user input is required besides a touch of a button, the "GUI" is a two-line LCD display, and the memory footprint of the system software needs to fit in under 256 kB?

On the other hand, programming even a lowly MP3 player could require hundreds of man-hours of development time if the developers are unfamiliar with the programming environment. Lean, microkernel-based OSes like QNX or VxWorks do not have large followings outside of the real-time or mission-critical embedded systems markets. They use many custom drivers and architectures that can be difficult to adapt for extended, unsanctioned, or otherwise nonstandard purposes.

The MASC system requires stability, flexibility, and familiarity in its runtime operating system. No operating system fits the bill entirely, but Linux comes close. On reasonably powerful platforms with modest amounts of memory, Linux can run very well with kernels ranging from near-micro to full-fledged. It is open-source, making the job of supporting MASC's admittedly strange approach toward harnessing hard-

[4]MP3 is a popular portable audio format. Handheld MP3 players are often slim, low-power devices which store one to two hours' worth of audio.

ware resources easier. It supports a wide variety of hardware—including the CardBus interface and the StrongARM CPU—through often cleanly portable interfaces. Perhaps most importantly, the Linux environment is very familiar to programmers, making the job of providing the right software functionality easier.

1.2 Thesis organization

Chapter 1 introduced the idea of the MASC system. Chapter 2 delves slightly deeper and discusses the current prototype implementation of the MASC hardware. The major contributions of this thesis were in the area of MASC software. Chapter 3 explores the kernel-level modifications made to enable proper software interfaces to the MASC hardware. Chapter 4 describes the user-space applications coded to demonstrate the capabilities of the MASC system. Chapter 5 describes the development process used to rapidly deploy these demonstrations. Chapter 6 summarizes the current state of MASC and discusses future applications of the system.

Chapter 2

The MASC prototype

The current MASC prototype system is built around an experimental hostcard and a development port of the Linux kernel. The PC Card format remains the primary standard interface through which the hostcard communicates, while Intel's StrongARM SA-1100 chip provides the actual computational power on the hostcard. The hostcard supports a variety of other interface protocols (described in further detail in Chapter 3) and its design does not preclude the use of other CPUs, but both choices are sensible for a first attempt at meeting the requirements of the MASC system. The first choice makes sense because a large body of PC Card peripherals already exists. The merits of the second choice are numerous: the StrongARM features excellent power dissipation characteristics, is a supported Linux architecture, and is commercially available in volume.

MASC software can be roughly divided along the lines of traditional operating system design. The kernel provides an abstract interface to the underlying hardware. Libraries access the kernel through system calls, and programs link with these libraries to provide interesting functionality (see Figure 2-2).

In a typical application of the prototype, an information appliance (IA) developer would permanently embed a MASC backplane into the IA and route access to the IA's electronics through an interface card or a CardBus bridge. When necessary, the end user would insert hostcards and peripherals into the backplane slots to add functionality to the information appliance. The hostcard could use a standard kernel

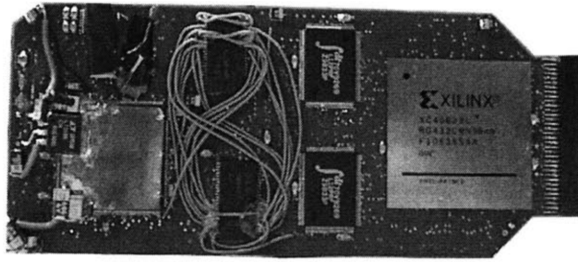


Figure 2-1: A prototype hostcard.

and libraries to provide basic features, but custom software could also periodically be downloaded onto the hostcard through a network peripheral.

2.1 MASC hostcard

The prototype MASC hostcard is depicted in Figure 2-1. The backplane connector is a 68-pin PC Card connector and is located on the right side of the card. The MASC hostcard can communicate using standard PCMCIA and CardBus protocols over the backplane bus. It does not meet the form factor requirements of the PC Card standard, but it remains very possible for this goal to be met with only minimal modifications to the design.

2.1.1 Specifications and features

Actual fabrication, and debugging of the prototype was documented in detail by Cheng [3]. The two most salient features of the prototype are the CPU, an Intel StrongARM SA-1100, and a large Xilinx XC4085XL FPGA[1]. The StrongARM, while not arbitrarily chosen, can still be considered a generic part which is not integral to the MASC system through its own virtues. The FPGA provides the core interface functionality. Other features include support for 16 or 32 MB of DRAM; a 4 MB flash EEPROM for booting and non-volatile storage; and 48 kB of dual-port RAM for FPGA I/O.

[1]Field-Programmable Gate Arrays (FPGAs) are programmable logic devices which can be used to prototype hardware very quickly through high-level languages such as VHDL (Vhsic Hardware Description Language).

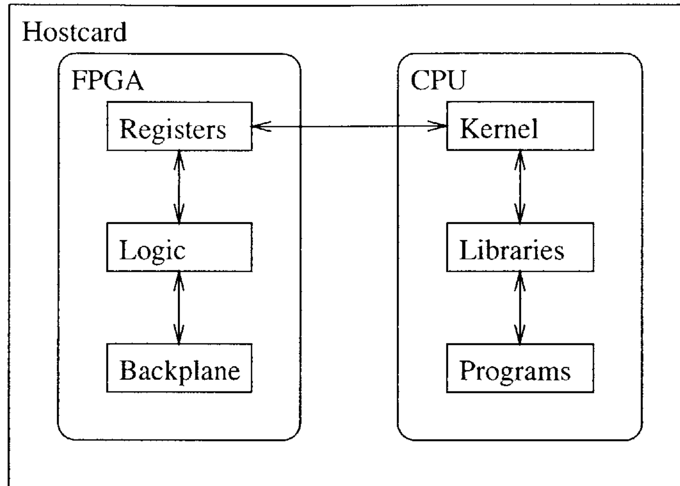


Figure 2-2: Block diagram of hostcard.

The MASC hostcard complies with the 16-bit PC Card and 32-bit CardBus protocols through the FPGA (see Figure 2-2). Like any PCI- or CardBus-enabled interface chip, the FPGA is capable of initiating PC Card transactions on behalf of the host and dealing with normal target events such as read requests, retries, and disconnections. Because only limited PC Card capabilities are required for this prototype, the FPGA does not present a standard PCI-to-CardBus bridge interface to the CPU. The actual register-level interface is described in Section 3.5.

For all intents and purposes, however, the FPGA does enable all of the functionality described in the PCI/CardBus specification. Through the FPGA, the hostcard is capable of configuring, controlling, and using any standard PCMCIA or CardBus card for which drivers are available. A few examples include network cards, storage cards, I/O enablers, and more.

2.2 MASC software

As described in the introduction, the MASC system employs Linux as the base of its software code.

2.2.1 ARM/Linux

Informally maintained by a group of volunteer developers, the ARM/Linux port is nevertheless quite stable and architecturally sound. Unlike most embeddable operating systems, ARM/Linux is not crippled, monolithic, or hopelessly hardware-dependent. The full capabilities of a standard Linux system are available (above the architecture level, of course) using ARM/Linux. This means that programs as high-level as X Windows, or drivers as low-level as the standard Linux PCI, IDE, and TCP/IP implementations[2] can both run with minimal to no modifications to their source code. All that is required below the architecture level is a compatible rewrite of I/O subsystem calls, a hardware-dependent task which would have to be implemented in some manner regardless of the operating system choice.

Neither is ARM/Linux monolithic. While the filesystem, main user interface, and application layer of operating systems like Windows CE are seamlessly integrated (theoretically), ARM/Linux is modular by design. It can use any standard filesystem for which drivers are available, including *ext2*, Windows VFAT, Windows NTFS, and NFS, to name a few. Like any UNIX system, its main user interface can be a shell like *bash* or *tcsh*, but a GUI based on X Windows or the General Graphics Interface (GGI) is also possible. Properly coded (in particular, portable) applications and libraries—graphics and audio codecs, for instance—can often be recompiled and run with no changes to their source.

The price paid for this power and flexibility is size and determinism. Linux is currently not as popular as QNX's QNX, Lynx's LynxOS, or Wind River's VxWorks in "hard" real-time applications (e.g., mission- or safety-critical systems with tight timing requirements) primarily because it cannot meet the binary size and latency requirements of most embedded applications [13]. There is some work being done, notably Cygnus' EL/IX, to "harden" Linux [10]. However, it is clear that until projects like EL/IX are completed, standard ARM/Linux may still prove valuable.

[2]PCI is the industry standard protocol for peripheral bus communication. IDE is the industry standard protocol for large disk access. TCP/IP is the *de facto* standard protocol for network communication.

As one real-time developer puts it, "Perfect execution may not be the most important criterion for open, consumer real-time systems. Rather, for nonsafety-critical systems, the right criterion is the most cost-effective execution" [13]. Linux offers both a familiar environment for writing new code and a standard interface for transparently porting legacy code.

Chapter 3

Kernel modifications and extensions

The run-time operating system environment is based upon a port of the Linux kernel to the ARM architecture. Although it is important to recognize that the MASC system is essentially processor-independent, ARM is still a sensible choice for a first implementation. ARM chips are available in both IP core and discrete-chip formats, have maintained a successful reputation for stability and low power dissipation, and are produced in volume by a variety of companies. To return to the point, ARM is also an excellent choice because it has been a supported Linux architecture since 1994 [8].

To provide the functionality demanded of a MASC hostcard, the ARM/Linux kernel had to be modified and extended to include support for the various firmware interfaces programmed into the FPGA. Minor changes had to be made to the kernel to tailor it to the hardware present on the prototype, and a number of custom kernel modules were produced to provide a proper software interface to the backplane bus.

Because of the wide variety of supported memory configurations, ARM/Linux currently cannot autodetect the memory layout of its host system. Hard-coding the layout before kernel compilation and/or letting the boot loader take care of passing in configuration parameters at run-time are both acceptable solutions.

3.1 Boot loader

Actually loading ARM/Linux onto the MASC hostcard can be accomplished in a variety of ways. The method favored by Cheng (Option 1 in Table A.1 and Figure B-1) employs DEC's[1] own "Angel" in-system debug monitor [3]. The majority of Angel is composed of hand-coded StrongARM assembly language routines. These routines enable, among other things, remote debugging over the StrongARM's serial port, complete control over processor state, and support for running arbitrary C, ARM, or Thumb[2] assembly code.

To boot ARM/Linux on the MASC hostcard with Angel, one first programs the onboard EEPROM with executable Angel code located at address 0 (on startup, the StrongARM begins executing code from address 0 of the EEPROM). Angel first initializes the memory subsystem of the board and proceeds to configure miscellaneous parameters (clock speed, DRAM timings, peripheral enable, etc.). It then enters a wait state and listens to the serial port for commands from a host computer running the ARM Software Development Toolkit (SDT). The most useful command calls an Angel routine that enables the host computer to download binary code to the StrongARM's main memory—for example, the ARM/Linux kernel. Once the kernel and an initial filesystem are downloaded, a final command is sent which jumps the StrongARM's program counter to the beginning of the ARM/Linux kernel, starts the Linux boot process, and effectively disables Angel for the remainder of the session.

Angel is invaluable for new hardware verification, early software design, and new kernel testing, yet for the purposes of booting and running Linux, only two of its routines are actually necessary: one to copy the kernel and initial filesystem from the host computer to the StrongARM, and one to jump the program counter to the beginning of the ARM/Linux kernel code. Thus, though Angel is designed to be a minimal debug agent, the majority of its routines are either redundant[3] or unnecessary

---

[1] Digital Equipment Corporation (DEC) produced the original StrongARM design before licensing it to Intel.

[2] Thumb is an alternative instruction set that is available on generic ARM cores such as the ARM7 or the ARM9 (the StrongARM does not support Thumb). [9]

[3] For example, Angel provides support for enabling the SA-1100 memory management unit

once the hardware has been verified and Linux is running. Additionally, there is no particular reason that the kernel and initial filesystem could not be stored directly on the onboard EEPROM, rather than copied over the serial port by the SDT.

Copying binary code and jumping the program counter are both trivial tasks that can be accomplished with a custom boot loader—Option 2 in Table A.1 and Figure B-2. The boot loader designed for the MASC hostcard was coded in ARM assembly and is extremely minimal. Its tasks include:

- Memory subsystem initialization: setting the type, size, and timing of the host-card DRAM. 21 lines.
- Serial port initialization: configuring the StrongARM serial port for Linux console login. 11 lines.
- General Purpose I/O (GPIO) initialization: configuring the GPIO lines of the StrongARM. Two of the GPIO lines provide clocks for the FPGA. The rest can be used as interrupt lines between the FPGA and the StrongARM or for debugging purposes. 19 lines.
- Entering superuser mode, clearing all interrupts, and setting the clock speed. 9 lines.
- Relocating the kernel and filesystem from EEPROM to main memory, and setting kernel command-line parameters. 21 lines.
- Jumping to the kernel. 1 line.

When compiled, the MASC boot loader weighs in at around half a kilobyte, versus Angel's 60 to 70 kB. Because the kernel and filesystem are stored locally in EEPROM instead of remotely on the host computer, they can be copied to main memory almost instantaneously compared to downloading. At a serial speed of 115200 bits per second,

(MMU), building a virtual memory map, and setting up page tables, yet ARM/Linux handles these tasks in its own way; in fact, one of the first steps taken by ARM/Linux is to disable the MMU temporarily.

a 600 kB kernel and a 2 MB (compressed) filesystem can take more than 3 minutes to download using Angel. In contrast, the boot loader's relocation loop uses 12 32-bit registers to copy data from ROM to RAM—more than sufficient to keep up continuous 8-word burst transactions to DRAM. The loop finishes in less than half a millisecond, and ARM/Linux itself can be up and running in less than 15 seconds.

3.2 Flash EEPROM driver

For many embedded applications, the 15 seconds from power on to ready state is an acceptable length of time. For other applications, the StrongARM provides many power management options, including suspend modes that allow the operating system to awaken and return to its previous running state much more quickly than rebooting, perhaps 5 to 10 seconds.

Because the MASC hostcard was designed to be portable and removable, however, it draws its power from the backplane of the information appliance, not from any internal source. Therefore, when it is inserted into an appliance, it must power on and boot up every time. Suspend modes are not available unless a battery is installed on the hostcard itself, an option that would be difficult to realize with the current form factor.

Since every insertion is thus a cold boot, the wait time would always be around 15 seconds if Option 2 (see Table A.1 and Figure B-2) were used. Surprisingly, the majority of the 15 seconds is spent decompressing the filesystem stored in EEPROM and copied to main memory by the boot loader. A minimal filesystem contains system libraries such as *glibc* (the GNU standard C library) and essential binaries such as *ash* (a tiny *bash*-like shell) and *busybox* (a collection of standard UNIX utilities) and can be as small as 900 kB when compressed. A more useful filesystem also contains user libraries, programs, and data—for example, a suite of audio codecs, players, and files—and can range from 1 to 3 MB in size when compressed (see Section 4.1 for details). After the filesystem is decompressed by the kernel using Option 2, it is placed in RAM and the standard *rd* (RAM disk) block driver is used to access it.

The RAM disk driver makes a block of RAM appear to be a (very fast) fixed drive at the expense of a chunk of main memory.

A large (1 to 3 MB compressed) filesystem can take up to 10 seconds to decompress and require 4 to 8 MB of main memory to use. A simple solution which both eliminates the decompression time and requires no additional main memory is to access an uncompressed filesystem stored directly on the EEPROM (Option 3). Options 1 and 2 both take the seemingly roundabout route of copying a compressed filesystem (either from the host computer or from the EEPROM), decompressing it to a reserved chunk of RAM, and ignoring the EEPROM thereafter. Option 3 simply treats the EEPROM as a fixed disk and accesses its data directly[4].

Obviously, not as much storage space is available if the filesystem is stored uncompressed on the EEPROM. Additionally, most EEPROMs, including the ones used by the MASC hostcard, are not designed for random access. Blocks of data can be written and erased only in very large (128 kB) chunks, and latency times are generally longer than those for DRAM. As a result, worst-case accesses (those spanning the natural 128 kB boundaries) can be orders of magnitude slower. With a simple cache system, however, the situation can be improved to the point of usability. An additional advantage of Option 3 is that user data is transparently retained from session to session. The only way to store data non-volatilely using Options 1 or 2 is to recompress the filesystem and write it back to the EEPROM before powering down.

A Linux block driver was written for the flash EEPROM used on the MASC hostcard (AMD AM29LV160BB). By writing certain values to specific addresses of the EEPROM, commands such as read block, erase block, and write block become available. The driver caches user-space writes to the current 128 kB block and writes the data back to the EEPROM via these commands before another block is accessed. The cache can also be flushed manually via the *sync* system call. The flash EEPROM driver, like most block drivers, only provides the low-level routines for reading and

[4]Theoretically, the kernel could also be accessed directly off of the EEPROM rather than relocated to main memory, but performance considerations dictate that the kernel should be stored in main memory while in use.

writing data to the device and knows nothing of the filesystem type or logical sector size being used. It is thus compatible with all of the standard Linux filesystem types such as *ext2*, Windows VFAT, and others.

Using Option 3, the time from power on to ready state decreases to 5 seconds. As for run-time performance, for most read-only applications such as streaming audio or video, the filesystem may never have to be modified and the difference between storing data in main memory and storing data directly on the EEPROM is negligible. For other applications, such as continuous data acquisition, flushing the EEPROM every 128 kB incurs a fairly heavy penalty. However, any serious application that would use the MASC hostcard and require more than a minimal amount of storage would probably employ an external storage system, such as a PCMCIA flash card or portable magneto-optical media. Regardless of the preferred boot option, the onboard EEPROMs were never intended for anything except storing the kernel and starting basic services.

3.3 LCD driver

The StrongARM SA-1100 includes an on-board liquid crystal display (LCD) controller that is capable of driving both passive and active LCDs with resolutions of up to 1024x1024 and color depths of up to 16 bits [4]. One of the few StrongARM-dependent drivers used by the MASC system (see Table A.2), the LCD driver provides standard support for graphical applications such as X Windows and the General Graphics Interface (GGI) through a frame buffer interface. A variety of LCDs can be used if minimal changes are made to the driver source, but a small, quarter-VGA (320x240) 4-bit LCD (Seiko G325E) was chosen as an example because of its small size.

All Linux kernels newer than version 2.2 include a standard linear frame buffer interface. Accessed through the device nodes */dev/fb**, the frame buffer devices are simply memory abstractions of underlying graphics hardware. A handful of *ioctl()* functions allow programs to change video modes, maintain palettes, and other miscellaneous functions, but otherwise, drawing on the display using the frame buffer

interface is as simple as setting bits in an allocated memory area. [14]

The SA-1100 ARM/Linux port partially supports the StrongARM LCD controller through the *sa1100fb* frame buffer driver—enough to drive a low-resolution LCD like the Seiko. The frame buffer driver was extended and lightly modified to support 4-bit palettes and to properly initialize the Seiko; otherwise, the code was left largely untouched.

Programming for the frame buffer is not difficult. Because it "presents a generic interface across all platforms" [2], programs and libraries that support the frame buffer as a graphics target are cleanly portable. The two major graphics libraries used by the MASC system are X Windows and GGI, both described in further detail in Section 4.2. The controller, driver, and libraries are not optimized or memory-efficient, but together they are perfectly capable of providing low-impact visual feedback for portable applications.

3.4 MASC driver

The need for modularity was recognized fairly early in the development of the MASC runtime system. Because patches for the ARM/Linux port are released quite frequently (often, ten to twenty times after every new release of the Linux kernel), keeping up with hardware and kernel version dependencies across a variety of custom drivers can become an impossible task. Restricting these dependencies to a single module outside of the kernel source tree is thus a necessary task for any serious development to take place. Additionally, since all of the MASC drivers besides the LCD driver communicate across the CardBus backplane, a single master driver should coordinate simultaneous transactions before placing them on the backplane bus through the FPGA.

The MASC global driver supports both of these functionalities. It dynamically provides to client drivers information such as base address locations of I/O registers, IRQ numbers, and buffer sizes. The client drivers are thus insulated from a large number of hardware conditions that can change across revisions of the FPGA code.

Before installation, client drivers such as the SPI, BPIO, EPP, and CardBus drivers first register their presence with and request resources (e.g., I/O and interrupt lines) from the MASC global driver through the exported function *masc_register_driver()*. If the requested resources are free, the function then marks the resources as in use and returns them in a structure. A more complicated design that supports dynamic real-time resource sharing through locks is also possible but unnecessary given the limited, 2-port backplanes currently being used with the MASC system.

The MASC global driver also activates and initializes the FPGA and handles system tasks such as resetting and entering low-power modes. It is the first module to be loaded and the last to be shut down when the hostcard is activated and powered off, respectively.

3.5 CardBus services

CardBus is a bus technology identical to PCI in its protocol details but intended for use with high-bandwidth removable peripherals. Like PCI, CardBus is 32-bit and runs at 33 MHz[5], but uses a completely different form factor and generally requires a combination of specialized hardware and software to support hot-pluggability.

CardBus's hardware requirements center around the need for portability, compatibility, and low power consumption. CardBus uses a compact 68-pin shielded connector that is identical to that of legacy 16-bit PC Cards (formerly PCMCIA cards) with the exception of a large number of extra ground pins. CardBus cards also share the same general shape of 16-bit PC Cards but are slightly thicker in most cases. To support removability, CardBus cards require a bridge to buffer transactions between the host computer's PCI bus and the CardBus bus. The PCI-to-CardBus bridge also manages the power consumption of any CardBus cards connected to it, a necessary function for portable applications which PCI does not support. [6]

Card Services and *Socket Services* are the recommended software APIs for pro-

[5]The PCI specification does not rule out 64-bit bus widths or 66 MHz bus speeds, but most common PCI implementations run 32-bit busses at 33 MHz.

gramming drivers for PC Cards (regardless of whether they are 16-bit or CardBus cards). Socket Services works on the bridge level and forms an abstraction of the bridge and socket hardware. It also manages power to cards and notifies the system of physical card events, such as insertion or removal. Card Services manages the configuration of and interaction with PC Cards once they are inserted. It queries cards and requests resources such as I/O base addresses and IRQ numbers on their behalf. [1]

Three types of transactions are available in the PCI/CardBus protocol to support a wide variety of functionality. Configuration transactions enable the host computer to access a PC Card's configuration address space, a 64-word space that contains, among other things, the card's identity, its resource requirements, and the standard Card Information Structure (CIS). The host computer first determines if the resources the card needs are available and, if so, allocates them and writes the results back to the configuration space. [1]

However, because PCI/CardBus devices need to be configured before they can even be seen by the operating system, a chicken-or-egg question often arises when developing Card Services implementations. The solution which most host computers employ is to let the computer's BIOS handle all configuration transactions in a hardware-dependent fashion. After a CardBus device has been configured, it can then be memory-mapped into the host computer's I/O or high memory address spaces and accessed normally by the operating system.

The standard Linux PCI configuration calls (*read_config_byte()*, *write_config_byte()*, etc.) were rerouted from the BIOS to the FPGA. The two other types of PCI/CardBus transactions—I/O and memory transactions—are handled by the Linux *inb()*, *outb()*, *readb()*, and *writeb()* calls, respectively. These, too, were modified to access the FPGA through a static memory map rather than through a configured PCI-to-CardBus bridge.

The Linux PCMCIA Card Services kernel modules requires few additional modifications to work with the MASC system. The current iteration of MASC CardBus support is considerably simpler and less extensible than standard implementations.

The job of porting Card Services was thus largely a straightforward task of removing unnecessary references to CardBus bridge hardware and adding hooks to access the FPGA.

Chapter 4

User space applications

4.1 Filesystem

A useful filesystem is as important to a runtime operating system as the kernel itself. Indeed, many in the community would have the phenomenon known as Linux renamed to "GNU/Linux" to reflect the fact that GNU, the Free Software Foundation's effort to create a complete UNIX-like system, has provided most of the tools, utilities, and libraries now in use on Linux systems. [12]

Especially on embedded systems such as MASC, the need for tight, efficient code is obvious. Heavily used system binaries such as filesystem utilities or dynamic libraries (against which dozens of other programs are linked), must be both small and fast to run on typical embedded processors. Because of the limited non-volatile storage available on the MASC hostcard (see Section 3.1 and the fact that the StrongARM is actually quite powerful compared to most embedded processors, size turns out to be more of a priority than speed for MASC.

A minimal filesystem for MASC includes the following:

- The GNU C library (*libc*). The C library must be present for any dynamically linked C language binary (practically every useful program) to run. 940 kB.
- Linker (*ld*). The linker library handles dynamic symbol resolution for linked programs. 90 kB.

- Miscellaneous libraries (*libcrypt, libdl, libnsl, libnss_files, libutil*). These libraries help to support login capability. 150 kB.

- Minimal POSIX commands (e.g., *cat, chown, cp, dd, kill, mount, rm*, etc.). These commands are useful for script-based automation of tasks. 68 kB.

- Shell (*ash*). The *ash* shell is a minimal *bash*-like shell with command-line history and scripting capability. 110 kB.

- System utilities (e.g., *getty, grep, init, login, ps*, etc.). These system utilities help to support login and task control capabilities. 150 kB.

- Filesystem utilities (e.g., *fsck, mke2fs*). These filesystem utilities maintain the integrity of *ext2* filesystems either in RAM or on the EEPROM. 30 kB.

Compressed, a snapshot of this filesystem occupies 850 kB. In contrast, the filesystem used by Cheng occupies approximately 4 MB [3]. The major savings occurred in the elimination of unnecessary libraries, and the use of multicall binaries developed by the Linux Router Project[1]. The *busybox* binary in particular provides an almost-complete set of POSIX commands, while a *sed*- and *ash*-based script provides the miscellaneous "system utilities."

All of these utilities were modified to cleanly cross-compile to the StrongARM.

4.2 Graphics library

Additional libraries may be added to the filesystem to provide more functionality. If an LCD panel is used in conjunction with the MASC hostcard (see Section 3.3), then graphics may be displayed by writing directly to the frame buffer device node or by linking to a standard graphics library such as the General Graphics Interface, or GGI. Many graphical applications, including variants of X Windows, many games, and the VNC virtual desktop protocol[2] support GGI as a target. With no modification to its

[1]See the Linux Router Project WWW page (`http://www.linuxrouter.org`) for details.
[2]See the VNC WWW page (`http://http://www.uk.research.att.com/vnc/`) for details.

source code, GGI and its adjunct libraries require 400-500 kB of additional storage space when compiled.

The most familiar graphics programming environment for UNIX users is X Windows. The MASC hostcard can support graphics programming via a number of options. The most direct option is to compile and run the frame buffer X server. The frame buffer X server is the least accelerated and least efficient of all of the X servers. When used in conjunction with even a minimal set of X libraries, it is also too large for practical consideration.

Another option is to run the GGI X server. There is no advantage in terms of speed or efficiency, as the frame buffer is still used as the eventual target. However, the low-level graphics display routines are more easily modified in the GGI source code tree than in the X source code tree. The last option is to run a tiny X-like server such as *nano-X* or the W window system. The latter is very small—about 200 kB when compiled—supports very basic windowing functions, and supports GGI as a target. It runs acceptably well, although it is far from extensible or professional-quality code. Still, W does provide a basic user interface and was ported with only a minimum of difficulty.

4.3 DAC library

The majority of the current demonstrations of the MASC system combine the hostcard, the backplane, and a peripheral based on a cheap digital-to-analog converter (DAC) with an SPI interface. A user-space library was designed and written to provide an abstract interface to the DAC. The library accesses the DAC through the SPI device node and exports several functions to programs that link with it. Thus, a variety of custom peripherals can be built around the same DAC, with code differences appearing only above the DAC library level.

4.3.1 Demo: MP3 player

One of the first demonstrations that employed the DAC was of an MP3 player. The MP3 decoding is done entirely in software and the results are written to output using the DAC library. Only one external chip is thus required for this peripheral to work.

The hostcard currently runs at about half of its rated maximum speed. With the hostcard running at 100 MHz, the MP3 decoder can process 56 kbps monoaural streams without any noticeable skips. It is expected that at full speed, the hostcard will be capable of CD-quality stereo sound using the same driver, library, and application code.

4.3.2 Demo: Motor control

The same DAC library was also used to control the motors and steering of a small toy car. The car's control system uses a simple protocol based on the timing between analog pulses to determine the speed of the rear wheel drive and the rotation of the front wheel steering column. The MASC hostcard is connected to the backplane and the backplane is mounted near the DAC and the car's electronics (see Figure 4-1). The bandwidth of the SPI interface is great enough to allow MP3 packets to be inserted in between car control packets. The result, shown in Figure 4-2, is an autonomous car with a digital stereo system.

4.4 ADC library

Analog-to-digital converters (ADCs) with SPI interfaces are also readily available parts around which interesting input devices can be built. With a minimal amount of glue logic, practically any kind of sensor—infrared (IR), encoder, touchpad, or accelerometer—can be turned into a MASC input device. The user-space library described earlier was extended to include complementary support for ADCs as well.

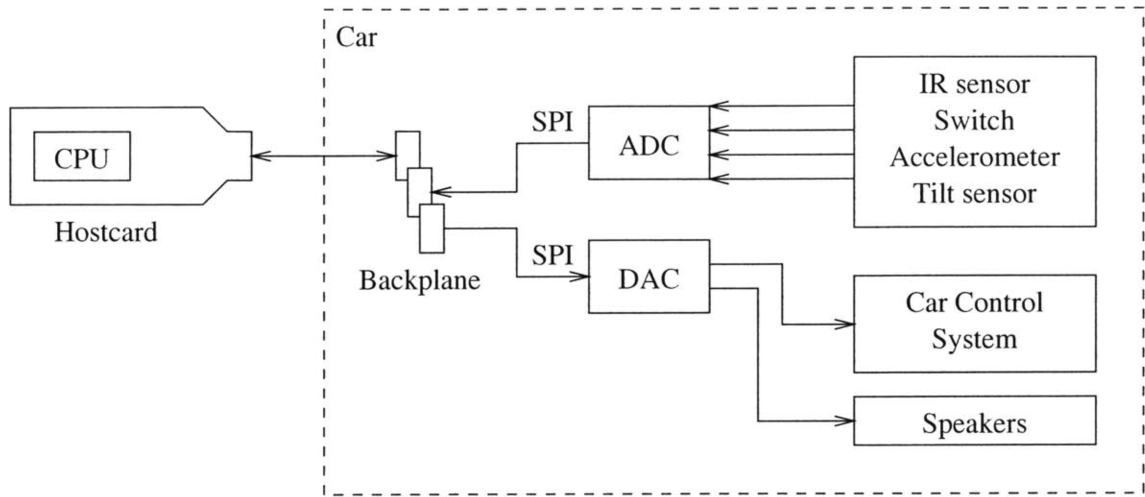


Figure 4-1: Car block diagram

Figure 4-2: Car

4.4.1 Demo: Sensor Control

The toy car described in Section 4.3.2 is effectively autonomous only if it is capable of obtaining feedback from its environment. Consequently, IR sensors, bump switches, and tilt sensors were added to the car soon after the hostcard was able to drive the motors. The sensors communicate with the hostcard through an ADC with an SPI interface. The ADC converts the analog signals created by the sensors and sends them in SPI format to the hostcard across the backplane. Software running on the hostcard takes the information presented by the sensors—i.e., position or tilt—and takes appropriate measures to prevent the car from falling off of cliffs or running into walls at high speed.

4.4.2 Demo: PDA

A simple personal digital assistant (PDA) was built to demonstrate a typical application involving the MASC hostcard, an SPI input device, and the LCD driver. The shell of the PDA contains a MASC backplane, an LCD panel, and an overlaid touchpad membrane (see Figure 4-3). The backplane's ports are exposed to allow for easy insertion of the hostcard and/or other peripherals. The idea is to separate the cheap standard components of a PDA (the display and the input device) from its intelligent components (the hostcard and other external peripherals). The PDA thus becomes completely modular: not only can its peripherals be easily upgraded and customized, but its "brains" can be as well.

In a typical scenario, the MASC hostcard plugs into the PDA shell and proceeds to drive the LCD and accept input from the touchpad. The process of driving the LCD is described in more detail in Section 3.3. The touchpad is connected to an ADC in the same manner as other sensors. Pressure applied to the touchpad creates a voltage that is converted by the ADC. A mouse driver running on the hostcard receives the SPI signal, interprets the position of the pressure, repackages the information, and sends it to a GUI displayed on the LCD.

As shown in Figure 4-3, the LCD interface bypasses the MASC backplane. While

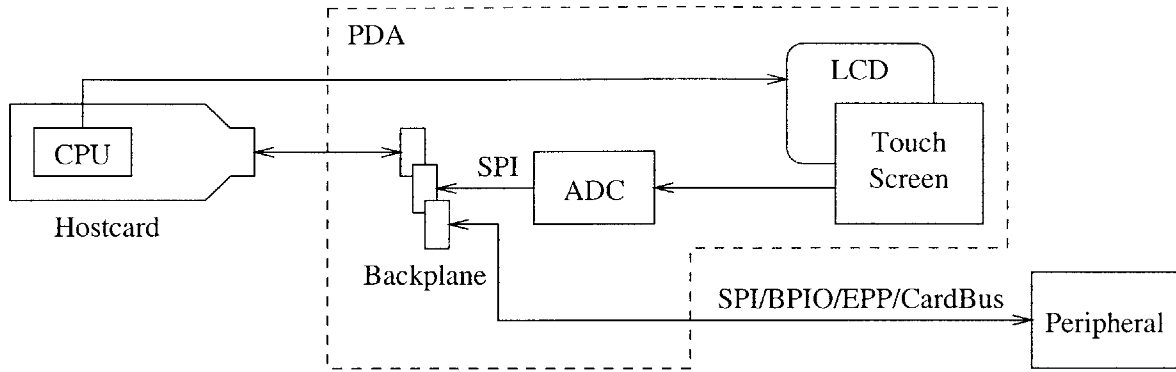


Figure 4-3: PDA block diagram

the LCD interface could theoretically be routed through the FPGA (making the modularity of the system much cleaner), the LCD's bandwidth requirements are great enough that a direct connection between the CPU and the LCD is almost mandatory. Future iterations of the MASC hostcard may include a standard auxiliary connector on the backplane bus for these types of miscellaneous interfaces.

Chapter 5

Development system and process

The MASC system as described would serve no useful purpose if it did not make the job of the IA developer easier as claimed in the introduction. Through the use of portable hostcards and peripherals rather than embedded CPUs, MASC enables developers to concentrate the majority of their efforts on using pre-built, pre-configured hardware in interesting ways.

In addition, MASC allows regular, existing appliances to be converted into information appliances with no additional modifications outside of implanting a backplane. The extensible backplane allows for creative, customized, and even unintended functionality by giving the user as well as the developer control of the appliance design.

Perhaps the most important benefit of the MASC system, however, is that most of the work of enabling information appliances is moved into software. The standard MASC platform allows software to catch up to Moore's Law in the IA domain, just as the standard PC platform has allowed in the PC domain.

Most of the MASC development process can take place on faster, more powerful, and more familiar systems than the eventual target (the IA). Cross-compilers and simulators allow programs to be thoroughly debugged and tested long before they are ever installed on the appliance. Finally, new MASC peripheral and interface design is made easier by the fact that the process can be pipelined into steps. Producing a MASC IA need not be a monolithic task.

5.1 ARM/Linux toolchain

The first step that must be taken before compilation of the kernel, libraries, and applications can take place is to establish an ARM/Linux "toolchain." With such a toolchain, code written in C, C++, assembly, and even Java can be developed on more familiar platforms such as the x86 or Solaris, compiled much more quickly, and debugged before being installed.

5.1.1 Cross compiler

The cross-compiler used for building everything from the kernel to the demos was the Experimental GNU Compilation System (*egcs*), which has recently been folded back into the *gcc* project. The methods used to build the cross-compiler are beyond the scope of this thesis, but the basic process involves

- Building a cross-compiling version of *binutils* to assemble code into object format.
- Building a cross-compiling version of *egcs* to compile C code into assembly.
- Building a native ARM version of *glibc*, the GNU C library .
- Installing and verifying each component of *glibc* on the hostcard.

5.1.2 Simulator

The simulator used for verifying the operation of compiled object code was the ARMulator, included in the ARM Software Development Kit. The ARMulator was invaluable for real-time testing of the boot loader especially. The ARMulator can both completely simulate the ARM CPU in software as well as remotely debug real hardware over the serial port via Angel (see Section 3.1). The ARMulator can provide register-level feedback from and control over the CPU.

5.2 Development process

To implement a new MASC interface or peripheral—for example, the EPP parallel port interface—four independent steps must be taken. First, the function of the device must be completely specified, either in an official white paper or through thorough discussion with both the hardware and software developers. Second, the actual logic must be laid out by the VHDL designer to meet the functional specification. The third step can occur simultaneously with the second. The driver designer must interface with the logic through a register-level description of the specification and create an appropriate abstraction of the hardware. Fourth, the application and user interface must be coded by the software designer to provide desired functionality.

This process can occur very rapidly if the hardware and software infrastructure is already laid down as in MASC. Programming a new driver for a CardBus peripheral can be a trivial task if the peripheral is well-designed and meets specification. Where creativity and the most interesting development takes place is in the application programming domain—specifying how the information appliance should use the new functionality.

Chapter 6

Conclusion

MASC's RTOS and familiar environment enable developers to shift much of the burden of producing useful, intelligent information appliances to software. The MASC approach does not accomplish this goal by forcing a computational standard upon the developer (i.e., by requiring the use of a specific hardware platform), but rather by offering computational power as a commodity—a scalable, modular, and generic unit of digital intelligence. The MASC RTOS can work with any combination of CPU and interface, including the StrongARM and CardBus, that supports this idea. As a result, adding intelligence to an IA can be as easy as plugging in a card, rather than worrying about embedding a particular CPU.

The development process itself does not have to involve any hardware programming if the supported standard interfaces—SPI, BPIO, EPP, or CardBus—are sufficient for the application. Of course, new interfaces can always be implemented as described in Section 5.2, but the BPIO interface is often sufficient for prototyping new protocols through its ability to control individual bits in software.

Application programming for MASC can be done in high-level languages such as C++ or Java to ease both portability and reliability. Modules programmed in these languages, such as audio or graphics libraries or demonstration programs, need not rely on particular underlying hardware implementations. This divorce is accomplished through the use of kernel drivers that hide these details behind standard interfaces.

Finally, the only limitation to MASC's extensibility in a particular system is the

number of ports on the backplane in use. Not only peripherals but also additional hostcards can be added at will. Parallel processing among several MASC hostcards can be accomplished through Beowulf-style clustering across the backplane[1]. If a network card is among the MASC peripherals on a particular backplane, drivers for the other peripherals may be dynamically updated as they become available. Control over the IA's functionality thus falls into the hands of the actual end user, rather than the initial developer.

[1]See `http://www.beowulf.org` for more information on the Beowulf Project.

Appendix A

Tables

Table A.1: Filesystem storage options

Option	Loader	Compressed	Stored on	Driver	Boot time
Option 1	Angel	Yes	Host	RAM disk	3 min
Option 2	Boot loader	Yes	EEPROM	RAM disk	15 s
Option 3	Boot loader	No	EEPROM	EEPROM	5 s

Table A.2: Driver dependencies

Driver	SA-1100	FPGA	Linux
Boot loader	Yes	No	No
Flash EEPROM	No	No	Yes
LCD	Yes	No	Yes
MASC	No	Yes	Yes
SPI	No	Yes	Yes
BPIO	No	Yes	Yes
EPP	No	Yes	Yes
CardBus	No	Yes	Yes

Appendix B

Figures

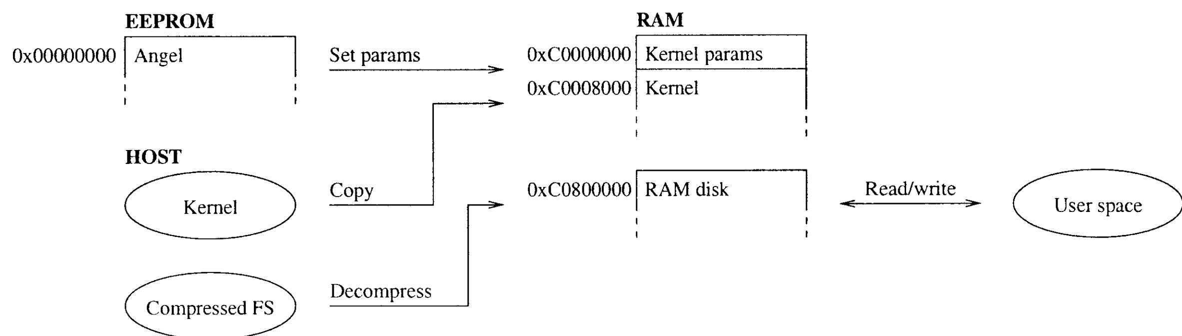


Figure B-1: Filesystem storage option 1

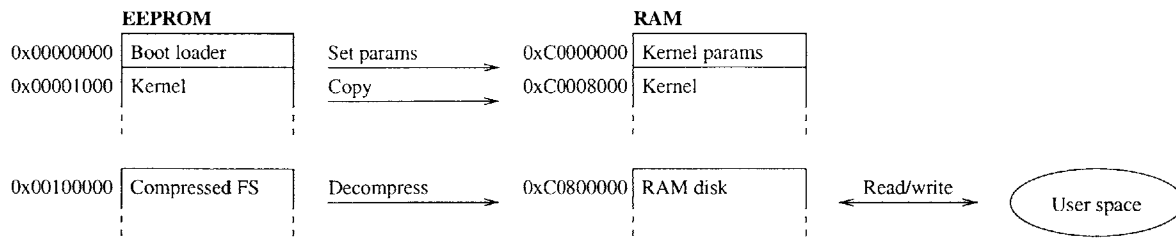


Figure B-2: Filesystem storage option 2

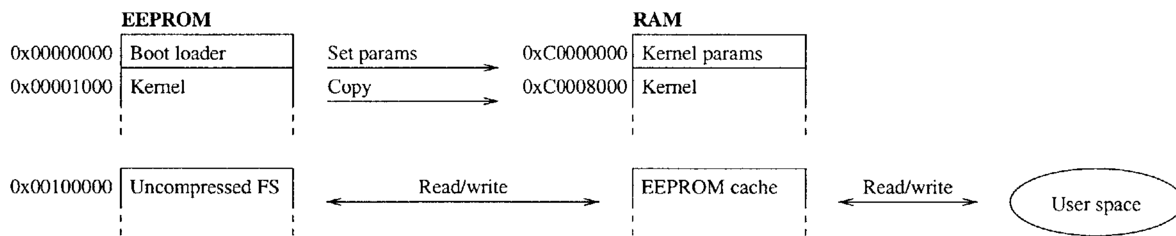


Figure B-3: Filesystem storage option 3

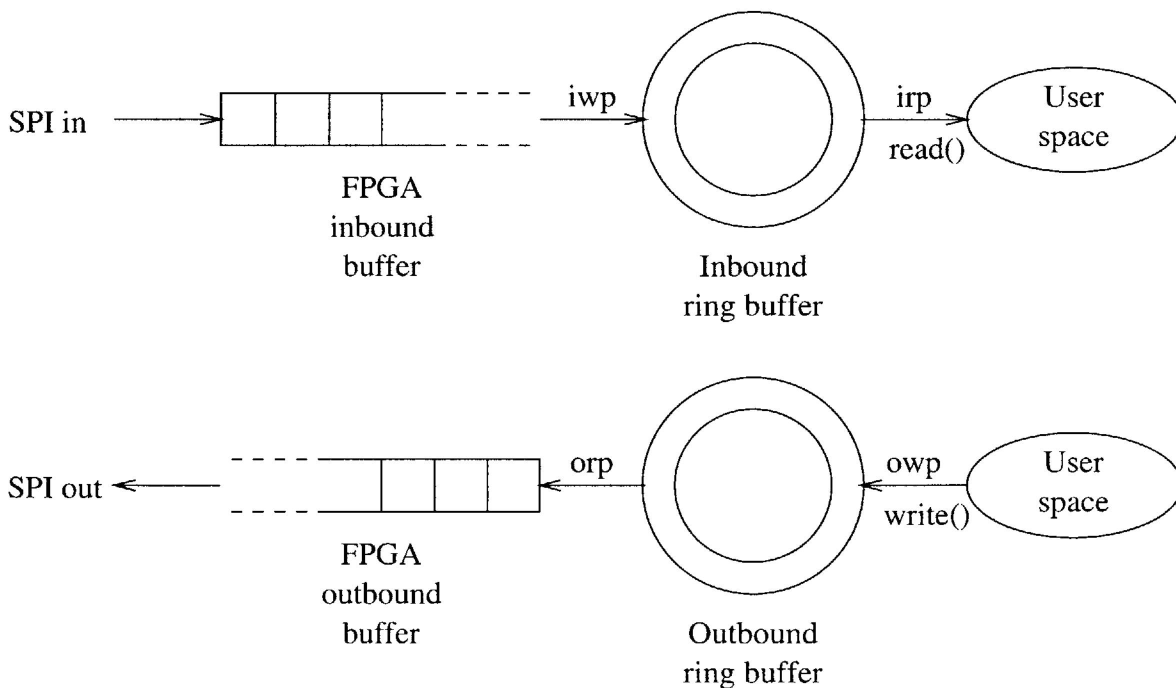


Figure B-4: SPI buffer system

Bibliography

[1] Don Anderson and Tom Shanley. *CardBus System Architecture*. PC System Architecture. Addison-Wesley, January 1996.

[2] Alex Buell. Framebuffer HOWTO. *http://linux.com/howto/Framebuffer-HOWTO.html*, July 1999.

[3] Cheng Cheng. Building the MASC information appliance prototype. Master's thesis, Massachusetts Institute of Technology, 1999.

[4] Intel Corporation. Intel StrongARM SA-1100 microprocessor developer's manual. Technical report, Intel Corporation, 1999.

[5] Intel Corporation. Pentium III processor datasheet. Technical report, Intel Corporation, 1999.

[6] Claude A. Cruz. New bus architectures: How CardBus fits with IEEE 1394, USB, PCI and others. Technical report, 1997.

[7] Arik Hesseldahl. Embedded MPUs seek the next killer app. *Electronic News*, pages 23–26, October 1999.

[8] Russell King. ARM Linux—the history. *http://www.arm.uk.linux.org/armlinux/history.html*, April 1998.

[9] ARM Ltd. ARM development systems FAQ. *http://www.arm.com/DevSupp/Sales+Support/faq.html*, 1999.

[10] Craig Matsumoto. Real-time Linux developers unite on API. *EE Times*, December 1999.

[11] Ephraim Schwartz. Intel gets serious about handhelds: juices StrongARM. *InfoWorld*, 21(18):8, May 1999.

[12] Richard M. Stallman. The GNU project. *http://www.gnu.org/gnu/thegnuproject.html*, 1998.

[13] John A. Stankovic. Strategic directions in real-time and embedded systems. *ACM Computing Surveys*, 28(4):751–763, December 1996.

[14] Geert Uytterhoeven. The frame buffer device. *Linux kernel source (linux/Documentation/fb/framebuffer.txt)*, November 1998.

[15] Alexander Wolfe and Junko Yoshida. Info appliances go prime time at Comdex. *EE Times*, November 1999.