Building the MASC Information Appliance Prototype

by

Cheng Cheng

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

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999

© Cheng Cheng, MCMXCIX. All rights reserved.

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

Author .................................

Department of Electrical Engineering and Computer Science

Certified by .........................

Srinivas Devadas
Professor

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

Arthur C. Smith
Chairman, Departmental Committee on Graduate Theses
Building the MASC Information Appliance Prototype

by

Cheng Cheng

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The MASC project is adopting a modular, networked approach to the design of
the next-generation information appliance. Our design separates the computation
resources from the appliance using computer cards and embedded backplanes. This
approach enables the transformation of any appliance into an intelligent, networked
information appliance. It also allows the appliance, the computer, and the software to
be upgraded independently. Our prototype computer card uses an Intel StrongARM-
1100 as CPU with 32 megabytes of DRAM and 4 megabytes of ROM. The computer
card communicates to the backplane through a customized I/O subsystem. Software
packages that have been ported to the prototype include the Angel debugger and
bootloader, Linux kernel 2.2, and Linux PCMCIA modules. By rewriting the kernel
I/O communication functions, existing Cardbus devices and device drivers can be
used on our prototype architecture. The simple application of a digital picture frame
demonstrates the capabilities of our infrastructure.

Thesis Supervisor: Srinivas Devadas
Title: Professor
Acknowledgments

I would like to thank a number of people whose ideas and help made this project possible. First, I would like to thank Prof. Srinivas Devadas for providing the guidance and the help throughout this project. I would also like to thank Sandeep Chatterjee for the ideas behind the MASC Information Appliance. Without his design, there would not be a MASC prototype. Additionally, his consistent help with circuit design and software building blocks were extremely valuable. I also want to thank Nicolas Nico, Phil Blundell, Russell King, and the other developers on the linux-sa1100 mailing list. They were a valuable resource in answering questions about the port of Linux to the StrongARM. Finally, I wish to thank my parents, friends, and other members of the CAA group for their support during this project.
Contents

1 Introduction .................................................. 11
  1.1 Information Appliances .................................. 11
  1.2 MASC .................................................. 13
  1.3 MASC Hardware Overview ................................ 14
  1.4 MASC Software Overview ................................ 16
  1.5 Contributions of This Thesis ............................ 17

2 System Software Infrastructure ............................... 19
  2.1 Debugger and Boot-loader ................................ 21
  2.2 Operating System ........................................ 24
    2.2.1 Arm-Linux Toolkit .................................. 26
    2.2.2 Porting Linux ........................................ 26
  2.3 Digital Picture Frame .................................... 28

3 Middleware Layers ............................................ 31
  3.1 PC Cardbus Implementation .............................. 32
  3.2 MASC Middleware ........................................ 35
  3.3 Middleware Performance .................................. 39
    3.3.1 Experimental Setup .................................. 39
    3.3.2 Performance Results .................................. 40
    3.3.3 Result Analysis ....................................... 42

4 Related Work ................................................ 45
## 4.1 VMEbus ........................................... 45
## 4.2 Jini ........................................... 46

## 5 Conclusion 49
5.1 Evaluation of Goals .................................. 49
5.2 Further Work ........................................ 50
  5.2.1 Hardware Improvements ............................... 51
  5.2.2 Software Improvements ................................ 52

## A Prototype Hardware 57
A.1 MASC Prototype Backplane ................................. 57
  A.1.1 Standalone Backplane ................................ 59
  A.1.2 Embeddable Backplane ................................ 59
A.2 MASC Prototype Hostcard ................................. 60
  A.2.1 Memory Subsystem .................................. 63
  A.2.2 I/O Subsystem ...................................... 64
List of Figures

1-1 MASC backplane and hostcard ........................................... 15
2-1 Angel debugger and boot-loader ........................................... 21
2-2 Complete digital picture frame .......................................... 28
2-3 Digital picture frame design .............................................. 30
3-1 PC cardbus infrastructure .................................................. 33
3-2 MASC middleware layer .................................................... 36
3-3 I/O data packet format ..................................................... 37
A-1 Backplane layout ............................................................. 58
A-2 Standalone backplane ....................................................... 59
A-3 Embeddable backplane ...................................................... 60
A-4 Hostcard layout ............................................................... 61
A-5 StrongARM-1100 CPU ....................................................... 62
A-6 MASC prototype hostcard .................................................. 64
List of Tables

2.1 Angel porting modifications .................................. 24
2.2 File modifications for Linux ................................. 27
3.1 Performance cost per function call ......................... 41
3.2 Middleware function call frequency on hostcard vs PC .... 41
3.3 Middleware cost with different protocols .................. 42
4.1 Industry software developments in automated networking . 47
Chapter 1

Introduction

As the new millennium approaches, the information age has clearly arrived. In this information age, anyone can have at his or her disposal a vast amount of information. Computers, cellular telephones, and pagers allow instant communication between parties anywhere in the world. Communication range, flexibility, and bandwidth are continuously growing. In this information age, automation of information services has become increasingly important. Although technological advancements have led to some degree of automation, there are still many manual tasks that are mundane and repetitive. This is especially true in the home. While businesses and government offices are increasingly using computers to automate many of the more tedious tasks, the information age has not yet arrived in the home. Building the automated home appropriate for the information age will require powerful information appliances.

1.1 Information Appliances

In the near future, home appliances will communicate with each other and automate many of the daily tasks in the home [5]. The appliances will cooperate to complete complex tasks and make useful suggestions to the user. Additionally, these appliances will be easy to use, modular, reconfigurable, and computationally powerful. To realize these information appliances, cheap, powerful, and reliable computing systems must be developed.
The networked Personal Computer (PC) can not be used in information appliances because of its complexity and lack of reliability. The personal computer is a large-scale computation resource. In home networks, most of the nodes in the network do not require a large amount of computing power. Additionally, a network environment also allows the computational resources to be non-local. Dedicating a PC to every appliance for home automation purposes would result in a large waste of resources. A single PC controlling all appliances over the network is also not a good solution for scalability and reliability reasons. It places a heavy burden on the network, restricts time-critical applications, and prevents the concurrent execution of multiple tasks. Thus, the PC, while useful for many applications, is not the computational cornerstone for home information appliances.

The embedded systems in current home products are inadequate for information appliances. While the current generation of embedded systems add functionality, they cannot be used for information appliances due to their limited computational resources and use of fixed components that are expensive to upgrade. Additionally, the embedded systems of current appliances work in isolation and are not intended to work in a networked environment. Thus, a new computing paradigm, that of information appliances, is necessary to take home automation to the next step.

Information appliances, or IAs, are “appliances” with computational resources and network awareness. While the name suggests only electronic appliances, IAs can pertain to any number of networked information processing devices, including televisions, toasters, CD players, garage door openers, and electricity meters. IAs differ from personal computers in their dependence on appliances. They are different from traditional appliances in the availability of greater computation power and network connectivity. Currently, there are numerous IA development efforts in industry and academia. However, given the breadth of possible information appliances, there have been few projects that have considered one complete hardware and software infrastructure standard for all such information appliances. This is where the MASC project begins.
1.2 MASC

The Modular Appliance Super Computer, or MASC, project at the MIT Laboratory for Computer Science has the goal of designing the infrastructure standard for the next-generation information appliance. The MASC project aims to design a hardware and software infrastructure that works well in a network environment and has sufficient computational resources to push the envelope in home automation. Additionally, the infrastructure must be simple to set up, easy to use, and completely foolproof. At the same time, the flexibility in the design should permit a breadth of uses and allow the end users to reconfigure and customize the computational structure to their needs. The MASC infrastructure should be easily adaptable to all appliances and enable the end users to dynamically build new ones. The following design goals, sorted by priority, were considered when designing an information appliance infrastructure.

- Most importantly, the information appliance infrastructure should be flexible for a variety of appliances. The same standard should be applicable to all appliances to minimize development cost and maximize user familiarity.

- The information appliance infrastructure should be user configurable. The user should have the power to build and reconfigure the computational resources to fit the computational requirements of each appliance. The reconfigurability should be in both hardware and software.

- Even with the power of reconfigurable hardware, the infrastructure should be easy to setup and easy to use. Information appliances should be simpler, more intuitive, and more robust than PCs.

- Information appliances should be upgradeable in both software and hardware. This is partly due to the difference in life cycles of consumer appliances and computation resources [4]. While a consumer appliance does not become obsolete for 5 to 10 years or more, computer resources, according to Moore's law, are often outdated in less than two years. Thus, the computation hardware should be upgradeable independent of the appliance.
• Information appliances should use home networking for information retrieval and software upgrades.

• Information appliances should be cheap so that they can be deployed in multiple appliances in a household.

• These information appliances should have sufficient computational resources to execute software applications such as AI, voice recognition, and real-time image processing.

• The computation resources, both within the information appliance and within the network domain, must be scalable.

• Finally, the computation resources of information appliances should work at low power settings to maximize battery life in a mobile setting.

Although these goals are all desirable, not all of them can be fully attained. In fact, some of these goals might be contradictory. The purpose and use of an information appliance then dictates the importance of these goals. Their relative importance can serve as a guide for the design tradeoffs in the MASC IA infrastructure.

1.3 MASC Hardware Overview

The design requirements in the previous section directed the design of the MASC information appliance infrastructure. The theme of modularity was deemed most important. By having a modular design, the goals of configurability, upgradability, scalability, and ease of use are satisfied. Additionally, a modular design allows one infrastructure to be adaptable to multiple appliances while minimizing the complexity to the user. Thus, we adopted a three-tiered modular approach to our design. The three tiers are the appliance, the computer, and the software [4]. The three tiers should be independent layers so that each can be changed separately. Then, one appliance can be upgraded with successive generations of new hardware and software.
Furthermore, our three-tiered modular approach allows users to customize and scale the amount of computation resources that are available to particular appliances.

The MASC infrastructure uses a bussed backplane and “smart” hostcard design. As seen in figure 1-1, the MASC hardware consists of a cheap, simple, and small backplane and a more expensive and complex computer hostcard. The backplane can be viewed simply as an arbitrated data bus. The computer hostcard contains all of the computation resources. To utilize the computational resources on the hostcard, the user simply plugs the card into the bussed backplane. Additional peripheral devices, such as hard drives, memory cards, and network cards, can also be plugged into the backplane. These devices can then be used by the hostcard as peripherals. The appliance is also connected via the backplane as a peripheral to the hostcard. Furthermore, multiple hostcards can be plugged into the same backplane bus to allow scalable parallel computing.

The modular design gives the user the ability to dynamically customize and configure the appliance and its computation resources. A user can plug in hostcards and different peripheral devices when appliances require different functionality. Furthermore, computation resources are easy to upgrade since hostcards can be easily
removed from the backplane. This ensures that the highest performance computational resources can always be connected to the appliance. This modularity also allows scalability of our design with multiple hostcards.

Network connections are easy to establish with our design. By using network cards connected to the backplane, a hostcard can communicate with file and program servers via the network to retrieve programs and data. This is an inexpensive and simple way to upgrade the software. Thus, our hardware infrastructure adopts the three-tiered approach and allows three degrees of freedom for system upgrades.

Based on this modular infrastructure design, we implemented the MASC hardware prototype. For the backplane, we chose the Cardbus standard, because it offers the performance of the PCI bus and the form-factor of PCMCIA. Additionally, adopting a commercial standard allows us to use a debugged system and the existing set of peripheral devices. The hostcard is comprised of two main systems, the computation resources and the backplane I/O subsystem. For the computation resources, we are using the StrongARM SA-1100 as the CPU. The StrongARM was chosen because it offers high performance, integrates many features onto the chip, is inexpensive, and requires very little power. The hostcard also has 32 MB of DRAM and 4 MB of ROM for storage purposes. The I/O subsystem is comprised of a Xilinx FPGA and three dual-port RAMs. These components provide the logic between the systems bus and the backplane. See Appendix A for more details on the design of the MASC hostcard prototype.

1.4 MASC Software Overview

Development of the MASC software infrastructure, like the development of the MASC hardware, was directed by the design goals listed in section 1.2. In order to maximize usability, and simplicity, the software infrastructure uses software modularity and incorporates automatic configuration of the hardware. The modular software approach minimizes development time while allowing the software to be reconfigured parallel to the MASC hardware. Incorporating hardware configuration through software enables
the user to reconfigure the information appliance dynamically. The infrastructure will support plug-and-play and allow hot-swapping of devices. With plug-and-play, the user can simply plug in a hostcard or peripheral device and it will be automatically configured. Hot-swapping allow devices to be removed and inserted into the backplane without power-cycling the entire appliance. With software automation and software modularity, user reconfiguration of the MASC infrastructure will be automatic, painless, and reliable. To minimize development time of the MASC software infrastructure, we port most of the system code from existing software. The key components of the MASC software infrastructure are the operating system and the middleware layer. Chapter 2 discusses the operating system in detail, and chapter 3 focuses on the middleware layer.

1.5 Contributions of This Thesis

For this thesis project, I worked on both the hardware and the software. On the hardware side, I implemented the designs for the prototype hardware. Specifically, I made several implementation design decisions for the prototype hardware. I also designed the prototype schematic and PCB layouts using the Accel Technologies TANGO PCB design suite. The focus of my thesis project is, however, on the software infrastructure of the MASC prototype, and in particular, the middleware layers. Chapter 2 focuses on the software infrastructure while chapter 3 discusses the middleware layer. For the software infrastructure, I specified and implemented the system software layers. This includes the structure of the software layers, porting the boot-loader and Linux kernel, and implementing the middleware layer. Finally, I implemented a digital picture frame using the hardware prototype and the software infrastructure.
Chapter 2

System Software Infrastructure

There are three main goals for developing the MASC systems software infrastructure. First, the software infrastructure must work seamlessly with the hardware. The software must work well with the reconfigurable, upgradeable, and modular features of the MASC hardware infrastructure. The best approach to satisfying this goal is to make the system software modular and reconfigurable as well. Secondly, the MASC system software must be simple, straightforward, and robust. The software layers should present a clean interface for applications making software applications easy to use and easy to develop. A clean and consistent development environment and system software are the first steps towards this goal. Finally, we wish to minimize development costs related to the system software. In order to rapidly develop the software infrastructure, we ported existing system software to the hostcard platform.

For the software infrastructure, software engineering techniques such as abstraction and modularity are used extensively. Due to the short time-frame within which the system software modules were implemented, we reused as much of the existing source code as possible. Code reuse also yields the benefits of complete software applications, development environment support, and networks of developers.

To simplify system software development, we take an incremental approach towards software development by using well-defined abstraction layers. This approach also leads to clean abstraction layers. Incremental development also shortens each development cycle. Each development cycle also gave quick feedback from run-time
results, which makes debugging code and adding new features much easier.

Our system software development strategy is comprised of five stages. The stages are:

1. **Hardware Debug** - in this first stage, simple StrongARM assembly codes test the hardware components and data paths. The memory and the I/O subsystems of the hostcard can be tested with comprehensive regression tests to determine whether the components and the data paths are working properly.

2. **Boot-loader** - now, a software debugger and boot-loader initializes the hostcard system and sets up the communication ports. At this stage, remote debugging of the system is available. This allows a much closer examination of the hardware. Additionally, simple compiled code can execute on the hostcard. The main goal of this stage is to set up the tools that reduce the development time of later software modules.

3. **Operating system** - at this stage, an operating system is ported to the hostcard system and loaded using the boot-loader. Careful consideration must be made in choosing the operating system (OS). A modular, micro-kernel based OS with multi-threaded, networking, and distributed computing support would be ideal, but development costs must also be considered. If the previous stage is completed successfully and we choose to port an existing operating system, this stage should be quick.

4. **Middleware** - at this stage, we add the software layer for backplane communication. This layer resides as modules independent of the operating system's core. The layer allows communication to other devices on the backplane. If possible, support for plug-and-play and automatic configuration will be added.

5. **Application** - at this point, the system software layers are working. Software development then focuses on the device drivers and applications that utilize the MASC infrastructure. Some possible software applications include digital picture frames, intelligent remote controls, and digital web phones [5].
The emphasis for the rest of this chapter will be on stages two and three of the software infrastructure. The boot-loader and debugger validates the operation of the hardware. The operating system provides a software foundation for applications. The middleware layer is discussed in chapter 3.

2.1 Debugger and Boot-loader

The system debugger and boot-loader are used for all the tasks mentioned in the previous section. It configures the CPU registers, loads device drivers for communication channels, and allows remote debugging of the hostcard hardware. Additionally, it allows downloading of applications and operating systems onto the hardware. There exists such a debugger and boot-loader for the StrongARM. It is called Angel.

Angel is a remote debug system that communicates with a host machine running the ARM Software Development Toolkit (SDT). Angel is also called the ARM debug agent. As seen in figure 2-1, communication between the host debugger and the remote Angel software utilizes the Angel Debug Protocol (ADP) [1]. The SDT can
run in both the Microsoft Windows NT environment and the SunOS environment. The Angel debug agent is comprised of only three components. The boot-loader component initializes the StrongARM CPU with the system architecture. The device drivers initialize and maintain communication to the host debugger. The Angel communication channels parse and execute ADP commands. Thus, Angel satisfies our requirements for a debugger and boot-loader software program.

Additionally, Angel is ideal for debugging and validating a new hardware architecture. Angel is, by design, extremely easy to port. As seen in table 2.1, all platform specific settings are stored together in a few files. Furthermore, Angel is small (about 60K in binary format), so it compiles quickly and can be easily loaded onto the hardware prototype. The small code-size also means fewer lines of code to debug. Angel can also be compiled in three versions for different stages of development:

- Full Angel Debug Agent - This is the first version loaded onto the prototype system since it supports the full ADP. It also supports application downloads onto the StrongARM platform.

- Minimal Agent - This removes all debugger functionally from the Angel code. Angel is then used purely to initialize a board and launch the appropriate application. This would be used in conjunction with the operating system.

- Late Startup Angel Code - Angel debugger is initialized only when the user application code encounters an error condition. This method is useful in debugging the operating system boot process.

The three methods allow flexibility in using the same Angel port for both kernel debugging and boot-loading the operating system.

Porting Angel is straightforward since the MASC hardware prototype is very similar to that of the Intel Brutus evaluation platform, which is already supported by Angel. Most of the modifications to Angel affect the initialization sequence of the ARM debug agent. The following list is a closer examination of the Angel boot sequence showing the locations where system initialization values are modified.
1. Upon boot, Angel switches to supervisor mode and sets the CPU clock rate to 206 MHz.

2. It then enables the DRAM and ROM banks as well as the GPIO and interrupt registers. These system values need to be modified to reflect the settings of the MASC hostcard prototype.

3. The debug agent then enables the memory-management unit (MMU) and creates a virtual memory map for the system [7]. The memory mapping also needs to be modified for MASC. On the hostcard, DRAM Bank 3 is remapped to become the FPGA in the I/O subsystems. The purpose of the virtual memory map is also to map the two discontinuous physical DRAM banks into one continuous block of virtual memory addresses. Additionally, the virtual address at 0x0 is remapped to DRAM so that applications can write exception vectors to that address.

4. Angel then loads the serial port device driver. It initializes the serial port and sets up the interrupt handler.

5. Finally, the ARM debug agent initializes the Angel Debug Protocol by setting up the communication channels. The ADP allows downloading and debugging of user application programs.

For our port of Angel, we started from an Angel port for the Intel Brutus evaluation board. We used the Brutus port due to the similarity between the two architectures. Most of the differences between Brutus and the MASC hostcard are in the memory configurations and the I/O subsystem settings. Table 2.1 lists all of the Angel files that were modified to port Angel for Brutus to Angel for the MASC hostcard.

With only these simple modifications to the Angel debugger, we ported the Angel debug/boot-loader environment to our hostcard. This allowed us to debug the hardware and allowed loading the operating system much more easily. Furthermore, Angel software can be developed in C or ARM assembly and downloaded to the hostcard in
seconds. Angel software can work closely with the hardware without worrying about access permissions or multi-threaded environments. Thus, it is ideal for developing appliance-dependent embedded applications.

### 2.2 Operating System

A number of operating systems were considered for the MASC information appliance infrastructure. Among the potential candidates were Java OS, Windows CE, Linux, NetBSD, Palm OS, and our own micro-kernel operating system design. We chose to use the Linux operating system for the following reasons:

- **Linux is free.** Linux is one of the few commercial operating systems which is free and whose source code is also distributed freely. Most other commercial operating systems, such as Windows CE, and Palm OS, do not have free source code, making prototype development expensive.

- **Linux is a stable operating system and not a specialized research operating system.** Extensive support is available. Furthermore, there are a large number of applications available for Linux. It also has a well supported graphical user interface in X-windows.

- **Linux has been developed extensively for the ARM architecture.** This is perhaps the most important factor since it greatly reduces the development time for

<table>
<thead>
<tr>
<th>Filename</th>
<th>Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>platform.s/platform.h</td>
<td>changed memory and ROM sizes, CPU register values, GPIO registers.</td>
</tr>
<tr>
<td>target.s</td>
<td>modified the memory map and boot-loader routines.</td>
</tr>
<tr>
<td>devconf.h</td>
<td>changed to IRQ instead of FIQ. relocated memory map location so it does not overlap the stack space.</td>
</tr>
<tr>
<td>mmu.h</td>
<td>changed the size of the memory map table</td>
</tr>
</tbody>
</table>

Table 2.1: Angel porting modifications
porting Linux to the prototype. Patches are available on the ARM Linux web page (http://www.arm.uk.linux.org/ rmk/) to port all major Linux versions to ARM and StrongARM based architectures.

- ARM-Linux has a strong developer base. This has proved to be an invaluable asset because the developer group has been very helpful in solving obscure operating system problems through mailing lists and discussion panels.

- Linux is a multitasking operating system. Furthermore, processes in Linux are queued and scheduled based on priority. Efficient scheduling and process switching are essential to maximizing performance with the hostcard I/O subsystem. Scheduling and context switching allow processes waiting for data from backplane devices yield the StrongARM microprocessor and allow other processes to execute.

- Linux has complete networking support. With networking support a part of the kernel, there are Linux device drivers for nearly all network devices. Additionally, network protocols and network stacks have already been built into the kernel. Linux also has PCMCIA modules which support plug-and-play for both PCMCIA and Cardbus cards.

- Linux has well-established tool chains. Standard compilers (gcc), linkers (ld), and debuggers (gdb) have ARM ports for the Linux operating system.

Of course, Linux is not the perfect solution. It is not the ideal operating system for the MASC infrastructure, because:

- Linux is not a micro-kernel based operating system. Ideally, we would like the MASC operating system to be modular with the core kernel less than 50 kilobytes in size. Additional OS components would then be loaded on demand. The monolithic Linux kernel is over 512 kilobytes. The large kernel size means more kernel code is loaded into memory. Fortunately, the Linux kernel can be trimmed by modifying the source tree.
• Linux does not have a small, compact graphical interface. Despite the availability of X-windows, Linux does not support a simple graphical interface for systems with very limited system resources. While it is possible to implement a new windowing interface with low resource requirements, this would dramatically increases the development cost to this project.

2.2.1 Arm-Linux Toolkit

The first step in porting Linux is to create the tool chain for building the kernel. Because of resource limitations in DRAM storage space, it was not possible to build a native Linux toolkit for the hostcard. Instead, we used a Pentium-II 300 MHz system to build the Linux kernel. Thus, we needed a cross-compile tool chain from the Pentium to the ARM microprocessor.

The gnu tool chain has the cross-compiler support for the ARM architecture. Additionally, both the aout and elf binary formats are supported. For the arm-linux-aout and arm-linux-elf tool chains, we built the gnu assembler, linker, and profiler from the binutils 2.9.1 package. The compiler was built from the EGCS 1.1.1 release of gcc. Additionally, glibc version 2.1 was built using the cross compiler and loaded onto the hostcard’s file system. This allows compiled binaries to be dynamically linked on the hostcard.

2.2.2 Porting Linux

The Linux operating system has been ported to the ARM microprocessor family. ARM platforms which have ports of the Linux operating system include EBSA, Acorn, and Brutus. Linux kernel version 2.2 is the most recent port of the Linux kernel. Again, the Brutus port is the closet ARM port of the Linux kernel to our hostcard architecture, because it supports the SA-1100 microprocessor. Therefore, the Brutus port is the starting point of our Linux porting process. To port Linux to the hostcard, the files in table 2.2 were modified. Once the system was reconfigured for the MASC hardware prototype, we were able to load the Linux kernel on our
### Table 2.2: File modifications for Linux

<table>
<thead>
<tr>
<th>Filename</th>
<th>Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>drivers/char/serial_sa1100.c</code></td>
<td>setup only one serial port on UART3</td>
</tr>
<tr>
<td><code>arch/arm/mm/mm - sa1100.c</code></td>
<td>modified the virtual memory page table so DRAM banks 0 and 1 have 16 MB of DRAM each, also added FPGA page support.</td>
</tr>
<tr>
<td><code>arch/arm/kernel/setup.c</code></td>
<td>changed the total size of memory to 32 MB.</td>
</tr>
<tr>
<td><code>arch/arm/kernel/head - armv.S</code></td>
<td>modified the boot sequence to allow self loading via the flash memory chips.</td>
</tr>
<tr>
<td><code>drivers/video/fbmem.c</code></td>
<td>added LCD device drivers for display.</td>
</tr>
</tbody>
</table>

hostcard. Furthermore, with the ARM-Linux tool chain, we are able to compile and execute any C program on the hostcard prototype.

On the MASC hostcard prototype, the Linux file system is stored separate from the kernel in a compressed ramdisk. This separation allows the file system to be modified independently of the system kernel. When booting, the kernel uncompresses the ramdisk and loads it as an ext2 type file system into memory. The operating system can then read and write to the ramdisk as if it were any storage device. The only difference is that the written data is not persistent. In order for the data to last when the hostcard is turned off, the ramdisk either needs to be stored into the flash memory banks or batteries can be added to continuously power the DRAM banks. For the hostcard, the uncompressed ramdisk has approximately 5 megabytes of storage space. It is approximately 1.5 megabytes in size when compressed. The Linux kernel and the ramdisk can both be stored on the flash memory or be downloaded through the serial port to the hostcard. Downloading both the kernel and the ramdisk to the hostcard is extremely useful during development, since it requires saves the time of programming the flash memory chips. This reduces the turn around time for developing the kernel and the ramdisk.

The ramdisk that we constructed for the hostcard comes mainly from the Compaq Itsy development efforts. The Itsy ramdisk has a minimal file hierarchy for Linux. It
has basic commands, system administration commands, and some character devices. However, the library files only supported aout binaries. We updated the ramdisk by adding library support for elf binaries with glibc version 2.1. Additionally, we added newer versions of the module commands such as `insmod` and `rmmod`.

### 2.3 Digital Picture Frame

With the hardware infrastructure and the low level system software completed, we built some simple applications to illustrate the power of the MASC information appliance infrastructure. One simple application for the MASC infrastructure is a digital picture frame. The digital picture frame has a liquid crystal display (LCD) panel embedded in the picture frame. The MASC backplane is also embedded in the picture frame. An end user simply plugs the hostcard into the backplane for the system to work. The system can load graphics software as well as different pictures. This configuration opens the door for a variety of user customizations. For the display software, the user can load screensavers, slide show software, or 3D rendering software. The pictures can also be loaded in different file formats including compressed formats. Programs on the hostcard can then uncompress and convert them before displaying.
Using our prototype hardware, we built a digital picture frame using almost no additional components (figure 2-2). The main display is a Seiko G325R01E300 quarter VGA monochrome liquid crystal display panel. It has a resolution of 320x240 pixels and a form factor of 60mm by 80mm. It is a low-power, FSTN LCD with either back light or reflective lighting. The LCD display has a connector interface that includes an array of bias voltages and the standard 4-bit data inputs. The bias voltages are supplied through a voltage divider and follower circuit [8]. The 4-bit data inputs are driven directly by the 4-bit monochrome LCD controller for the SA-1100. Additionally, the SA-1100’s LCD controller is also driving the frame clock, pixel clock, line clock, and the AC bias clock of the LCD (figure 2-3).

In testing, this configuration requires a very low amount of power. Running on three C-sized batteries, the entire system works at 4.5 volts and only 0.11 amps. This means that the three batteries can sustain the digital picture frame for over a day of continuous operation. Adding sleep mode to both the hostcard and the LCD panel can extend the battery life. Although the current digital picture frame requires some additional wiring, future versions of the digital picture frame will not. Furthermore, using the SA-1100 to drive the LCD panel incurs a burden on the CPU and the system bus, because the LCD controller is continuously refreshing the picture and loading the frame buffer from DRAM. This performance cost can be circumvented by sending the pictures through the backplane to another frame buffer embedded in the picture frame. A separate LCD controller is also needed on the picture frame to directly drive that frame buffer. While this solution is more expensive by adding components to the backplane, it also reduces the data traffic on the system bus of the hostcard and frees up more computation resources for tasks such as data decompression or graphics algorithms.

For our prototype digital picture frame, we converted the graphics format of the images for drawing the pictures into the frame buffer. The program on our hostcard system converts the picture from a standard 256-color gray-scale bitmap to the 16-color format used in the SA-1100 frame buffer. Once the data is converted from normal bitmap format to the SA-1100 frame buffer format, it is stored in the DRAM.
frame buffer on the hostcard. The LCD shows the new picture whenever the SA-1100 LCD controller refreshes it. Since each bitmap picture takes less than 39 kilobytes 1, the prototype digital picture frame can easily store 25 bitmap pictures in less than 1 megabyte of flash memory. With four megabytes of flash memory, almost 100 bitmap pictures can be stored for our picture display. Of course, more pictures can be stored using compressed formats. Furthermore, a memory card can also be added to increase the storage space for pictures. With the potential for network connections, the hostcard can fetch pictures over the network and display them on the picture frame.

The digital picture frame clearly demonstrated the potential of the MASC information appliance infrastructure. With MASC, significant computational resources can be provided to previously static objects. The user also has the capability to reconfigure the computation structure of the system as needed. The functionality of a picture frame has exploded with the MASC infrastructure.

\[320 \times 240 \times 1/2 \text{byte per pixel} + 4 \text{ bytes} \times 16 \text{ colors} + 54 \text{ byte header} = 38,518 \text{ bytes per bitmap file}\]
Chapter 3

Middleware Layers

For our software infrastructure to take advantage of the MASC hardware, it must communicate with other devices on the backplane. This feature can not be directly ported from another system, since the hardware infrastructure of MASC is unique. Thus, a new software layer must be implemented to support this feature. We call this software layer the middleware layer. The middleware layer has the primary role of allowing communication with other devices on the backplane. A clean implementation of this software layer is essential, since nearly all MASC-based applications depend on it. For usability purposes, we want this software layer to allow support for automatic configuration and initialization of devices. Finally, we want to minimize software development time to reach a stable environment. In order to reach this goal, we want our middleware layer to be backward-compatible with existing applications, device drivers, and configuration services. That means support for existing device drivers so no modification of the driver code is necessary. In order to reach these objectives, a careful and clever implementation must be used.

We made some crucial design decisions in order to facilitate our implementation of the MASC middleware layer. First, we decided that the middleware layer will be integrated at the kernel level so that applications and device drivers can be abstracted away from these low-level system commands. Second, the software components for backplane communication should reside as kernel modules. Linux kernels version 2.0 and later have module support, so that software modules can be loaded dynamically.
Dynamically loaded modules not only add flexibility and reduce debug time but also introduce clean abstractions. The added bonus of clean abstractions means the kernel can be modularly reconfigured parallel to the reconfiguration of the hardware components. For the purposes of minimizing development time and having the data structures for automatic device configuration, it was only logical to port portions of the Linux PCMCIA modules to use our middleware layer. The PCMCIA code already supports plug-and-play, hot-swapping, and other automatic configuration features. Thus, we can reuse that portion of PCMCIA software, but modify it to support the MASC hardware infrastructure. In order to use the existing PCMCIA code and device drivers, a closer examination of implementation of the PCMCIA software modules is required.

3.1 PC Cardbus Implementation

On the PC, the PCMCIA and Cardbus devices are connected via a Cardbus bridge to the PCI bus of the host system. As seen in figure 3-1, all PC card device requests travel via the PCI bus and the Cardbus bridge to the PC card device. The Cardbus bridge is a hardware component that bridges the PCI bus and the Cardbus bus that actually communicates to the PC cards. The Cardbus bridge provides an abstraction barrier by masking the devices on the Cardbus bus as PCI devices. In order to maintain this abstraction layer, the Cardbus bridge stores the memory and I/O maps which remap the PCI addresses to Cardbus addresses and vice versa. Additionally, it reroutes the interrupts from the PC cards to PCI interrupts. Thus, to the system, the PC cards behave like PCI devices. The Cardbus bridge also supports additional functions such as PC card status updates, socket resets, and socket power cycling. The Cardbus bridge can be configured like a PCI device. Some configuration registers are used to configure the PC cards connected to the bridge. Thus, the Cardbus bridge is essential in a PC architecture for configuring and communicating to the PC cards.

On the software side, the PCMCIA and Cardbus system is divided into configuration software layers and run-time software. The configuration software layers deal
Figure 3-1: PC cardbus infrastructure
with setting up the device and configuring the host system for the device. The run-
time software includes the applications and the operating system which utilizes the
device through device drivers. For the configuration software layers, the Socket Ser-

dices layer first sets up the socket by resetting the socket and powering the socket to the

current voltage specified by the SOCKET_HOT_RESET register [3]. Then, Socket Services set
the IRQ mask and the IRQ value for the individual sockets. Card Services then reads
the Card Information System (CIS) data from the card [6]. Based on these values,
Card Services allocates the system resources necessary for the card and configures the
PC Card via the Cardbus bridge. During this step, Card Services remaps the I/O

and memory space of the PC card into PCI address space. The mapping boundaries
are stored in memory map and I/O map registers within the Cardbus bridge. These
values are also updated on the PC Cards. After the card device has been setup,
Card Services load and initialize the corresponding device driver for the PC card.

After these configuration steps, the applications and device drivers can then com-
municate to the PC card device through the system remapped address space. The
Linux PCMCIA modules contain faithful implementations of both Socket Services

and Card Services. Additionally, The modules support both PCMCIA and Cardbus
devices and device drivers.

Once the correct device driver is loaded via Card Services, the applications and
the operating system can utilize the device. While it is possible for the application to
directly access the Cardbus hardware, most device requests go through the operating
system and the device drivers. All device requests sent by the device driver are routed
through the PCI bus and passed through the Cardbus bridge. Interrupts are routed
to the interrupt handlers in the device drivers. These are essential for direct memory
access (DMA) transfers. There are standard Linux PCMCIA and Cardbus device
drivers for most common peripheral devices.

Although the entire process seems very complicated, it is not. The Cardbus bridge

provides a hardware abstraction from the PC card to the PCI bus. Above that, Socket

Services and Card Services provide software abstractions in memory and I/O maps for
setting up the device drivers. These services layers do the dirty work of manipulating
the sockets and setting up the resources so that the device drivers can be implemented in a straightforward manner.

3.2 MASC Middleware

As seen in figure 3-2, the MASC hardware infrastructure does not support a Cardbus bridge. Additionally, there is no PCI bus. Instead, the I/O subsystem consisting of a Xilinx FPGA serving as an interface between the system bus and the Cardbus backplane. Thus, all backplane device requests must be routed through the FPGA to the backplane.

The middleware layer routes all software requests between the system bus and the backplane through the FPGA. Given that Linux has good implementations of PCMCIA services modules and device drivers, a very low level software layer is optimal. This allows maximum backward compatibility with minimum code change.

In order to support backplane communication, the middleware layer is implemented as a well-defined interface for the PCMCIA configuration services and device drivers. As seen in figure 3-2, the middleware layer reroutes and modifies all backplane related requests to the FPGA. The middleware layer can almost be viewed as a very low-level device driver for the hostcard’s I/O subsystem. It is like a wrapper around the I/O subsystem. To the existing software modules, drivers, and applications, the hardware behaves very similar to that of the PCI bus on a PC.

The specific implementation goal of the middleware layer is to provide a data access interface identical to that of Linux on the PC. Doing so ensures that nearly all existing device drivers can work on the MASC hardware infrastructure without any code modifications. This flexibility comes at a cost of lower performance and higher data latency (section 3.3).

The implementation details of the middleware layer are fairly straightforward. The functions of the middleware layer work very closely with the VHDL code of the FPGA on the hostcard I/O subsystem. On our prototype, the I/O subsystem can handle one of four types of data accesses: an I/O write packet, an I/O read packet,
Hardware

Host System
(CPU + Memory System)

System Bus

PC Card

Backplane

XC4063

Software

Configuration and Setup Software

Card Services

Socket Services

MASC Middleware

RunTime Software

Applications

Operating System

Device Drivers

Figure 3-2: MASC middleware layer
a DMA write packet, and a DMA read packet. The FPGA converts each of these data requests into the appropriate backplane signals or drives the appropriate values on the system bus. Each of the data packets are four doublewords long. As seen in figure 3-3, each of the I/O data packets have all the necessary addressing information to retrieve or send the data to the device on the backplane. The value of the byte enable (BE) determines whether the packet is a byte, word and doubleword data accesses. The CC value determines the type of data access. Possible types include I/O, memory, and configuration space. All of the backplane device’s address space can then be read or written. Thus, the middleware layer has the task of repackaging all device data requests into one of these four packets. The following Linux kernel requests are trapped and handled by the MASC middleware layer:

- Configuration space functions: config_readb, config_readw, config_readl, config_writeb, config_writew, config_writel
- I/O space functions: inb, inw, inl, outb, outw, outl
- Memory space functions: readb, readw, readl, writeb, writew, writel
- DMA function: request_dma.

When the middleware layer is called by these functions, it first remaps the system address into the appropriate backplane address. It then calculates the byte enable and CC values. Finally, it sends the 16 byte data packet request to the FPGA which is mapped as DRAM Bank 3. In the case of read accesses, the middleware layer first writes a read request onto the backplane, and then stores the device response into a
register buffer. The middleware layer function can then retrieve the data as if it was a memory read.

Additionally, the middleware layer reroutes device interrupts to the appropriate interrupt handler. Because all devices on the backplane share one generic interrupt, there must be a generic interrupt handler to handle all backplane interrupts. If any of the backplane devices interrupt the hostcard, then the FPGA interrupts the CPU. At this point, the middleware interrupt handler is called. The interrupt handler then looks at the FIQ and IRQ interrupt registers within the FPGA. By reading these registers, the interrupting socket and device can be determined. The appropriate interrupt handler for that device is then executed.

In order to automate the configuration process in the infrastructure, we also ported the PCMCIA Socket Services and Card Services to the prototype. As described in section 3.1, Socket Services have the main task of programming the Cardbus bridge. In our hardware infrastructure, we do not have a Cardbus bridge to program (figure 3-2). However, we do have some backplane logic that can be programmed for rudimentary socket controls. Without a Cardbus bridge and some functionalities of the backplane, the capabilities of Socket Services has been greatly reduced. In the MASC infrastructure, Socket Services have very little real functionality. It mostly serves as skeleton code for maintaining backward compatibility.

As described in the previous section, Card Services is a collection of service functions for use by device drivers. It simplifies the task of writing device drivers, reduces resource conflicts, and allocates the system resources of the PC Card devices. In some ways, it is the central software component that allows plug-and-play. Surprisingly, given the complexity of Card Services, a very small amount of its code has to be modified, because it is well-implemented and deals mostly with device driver software. By routing all device data requests through the middleware layer, Card Services is abstracted away from the difference in the hardware infrastructures. Along these same lines, most device drivers that we have ported to the MASC hostcard require minimal code modification to work. Thus, the middleware layer achieves its goal of abstracting the existing code from the MASC infrastructure while allowing software
automation and preserving the modularity of the software infrastructure.

3.3 Middleware Performance

The middleware layer creates a clean abstraction for existing configuration software and device drivers. While it minimizes development time, allows automatic configuration, and preserves software modularity, the middleware layer does incur a performance cost. This cost comes from the extra instructions necessary to repackage the I/O and memory requests into the format used by the FPGA. The purpose of this section is to give a quantitative result on the performance cost of the middleware layer. These tests focus on whether the performance cost is significant enough to warrant a different implementation of the middleware layer.

3.3.1 Experimental Setup

Ideally, we would like to run a complete black-box test on the MASC hardware and software infrastructure. We would run a sequence of device intensive applications on the prototype and measure its performance. Then, we would measure the cost of MASC middleware layer by calculating the cost per middleware layer function call and the number of such calls. The total cost of the MASC middleware layer would be the product of the two. The real penalty of the middleware layer can then be calculated as:

\[
\text{cost of middleware layer} = \frac{(\text{time spent per call}) \times (\# \text{ of calls in test})}{\text{total time of test}} \tag{3.1}
\]

However, measuring the cost this way is not possible. At the time that this thesis was written, the hostcard could not communicate fully with devices on the backplane. The problem is due to the incomplete VHDL program for the XC4062 FPGA. The hostcard can not route all data requests onto the backplane. DMA requests are completely unsupported. Thus, there is no way to communicate with
other backplane devices. It would not be possible to setup a complete network stack and test the true performance of this system. Thus, a black-box testing methodology will not work. Rather, a somewhat less rigid approach is used by using a different architecture platform to obtain certain values. In the tests, we used a Pentium-90 PC with the same version of the Linux kernel to obtain the values for the number of function calls. The cost of each middleware layer function call is obtained using the MASC prototype hostcard. Together, these two values give an upper bound estimate on the cost of the middleware layer. We believe that this testing method is valid, because the measurement on the Pentium is the number of function calls. This value should not be platform-dependent. By using the same kernel source, the compiled network stack and driver calls are identical. In other words, with the same source code for network protocols and network stacks, the number of I/O functions called by an application will not change dramatically between the two architectures. Thus, obtaining the number of middleware function calls on a PC allows us to calculate the cost of the MASC middleware layer on ARM. We can then observe whether the middleware layer adds an unacceptable amount of performance cost.

3.3.2 Performance Results

We first measure the cost related to each middleware layer function call. The values are calculated by measuring the time required to make 300,000 calls to each middleware function. Then, we measured the time to make 300,000 calls to "fake" middleware functions. These "fake" middleware functions are direct FPGA accesses without repackaging of the data requests. By finding the difference between the two values and dividing it by the number of function calls, we obtain the values in table 3.1. The results are categorized into six types of data accesses, because the performance costs of byte, word, and doubleword accesses nearly identical. Data size does not affect the performance cost because all backplane data accesses have to be packaged into the 4 doubleword packet required by the FPGA.

Once we are able to determine the cost of each middleware layer function call, we need to calculate the number of function calls. As discussed in the previous section,
|
|---|---|
|Type of Access| Cost per function call|
|Configuration writes| 4.2uSec/call|
|Configuration reads| 8.1uSec/call|
|I/O writes| 3.2uSec/call|
|I/O reads| 6.8uSec/call|
|Memory space writes| 3.5uSec/call|
|Memory space reads| 7.2uSec/call|

Table 3.1: **Performance cost per function call**

<table>
<thead>
<tr>
<th>number of packets</th>
<th>PC In calls</th>
<th>PC Out calls</th>
<th>MASC In calls</th>
<th>MASC Out calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>863</td>
<td>27564</td>
<td>650</td>
<td>27445</td>
</tr>
<tr>
<td>1000</td>
<td>3847</td>
<td>274354</td>
<td>3542</td>
<td>272207</td>
</tr>
<tr>
<td>10000</td>
<td>32545</td>
<td>2732123</td>
<td>31234</td>
<td>2720898</td>
</tr>
</tbody>
</table>

Table 3.2: **Middleware function call frequency on hostcard vs PC**

we can not measure this on the hostcard prototype. However, because the kernel code is identical on both platforms, the number of function calls should be approximately the same on both the StrongARM hostcard and on the PC. In order to verify this hypothesis, we implemented several socket tests which simply dump data through the network stack. As seen in table 3.2, the number of middleware function calls are approximately the same for larger quantities of data transmission. The variation in the number of function calls can be attributed to the difference in loading network services. Thus, for high throughput tests, the PC middleware call frequency can be used as an approximation of the function call frequency on the StrongARM.

On the PC, we used a variety of protocols to test the frequency that the middleware layer is accessed. We ignored testing configuration data accesses because they occur infrequently and are only used during device initialization and setup. We also did not test memory reads and writes since it was difficult to setup such a testing environment with PC applications. Thus, we tested only I/O accesses to the middleware layer, because they are one of the more demanding types of accesses in terms of high throughput and low latency. Furthermore, we tested network performance
The test setup used two PCs connected via a cross-over Ethernet cable. The network device driver was modified to include a counter for middleware layer function calls. We booted one of the machines with the modified Linux kernel and executed the following benchmark tests across the network. We used ping and sshd as low throughput tests. To test the worst case for the middleware layer, we tried several high throughput, large data transfer tests. Specifically, we tested the ftp and http protocols. With these protocols, we generated a random set of 1000 web-page sized files. Then we tested the performance of the system by retrieving or sending all 1000 files using either ftp or http. The results of these tests are in Table 3.3. The P-90 Benchmark Execution Time column lists the time it took for the benchmark to execute on the Pentium 90. It should not be used to correlate with the performance of the StrongARM hostcard.

### Table 3.3: Middleware cost with different protocols

<table>
<thead>
<tr>
<th>Network Protocol</th>
<th>Input Access</th>
<th>Output Access</th>
<th>Middleware cost</th>
<th>P-90 Benchmark Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ping outbound</td>
<td>26555</td>
<td>32890</td>
<td>0.4sec</td>
<td>16min 30sec</td>
</tr>
<tr>
<td>sshd server</td>
<td>1054590</td>
<td>1669661</td>
<td>12.5 sec</td>
<td>18min 23sec</td>
</tr>
<tr>
<td>ftp upload</td>
<td>497762</td>
<td>8362113</td>
<td>30.1sec</td>
<td>3min 40sec</td>
</tr>
<tr>
<td>ftp download</td>
<td>8313173</td>
<td>264453</td>
<td>57.4 sec</td>
<td>3min 35sec</td>
</tr>
<tr>
<td>http server</td>
<td>445101</td>
<td>8361556</td>
<td>29.8sec</td>
<td>3min 25sec</td>
</tr>
<tr>
<td>http request</td>
<td>8318885</td>
<td>247086</td>
<td>57.3sec</td>
<td>3min 21sec</td>
</tr>
</tbody>
</table>

because it is the most important type of I/O operation for the MASC infrastructure.

3.3.3 Result Analysis

In Table 3.1, the cost of each function call is approximately the expected values. The read accesses take approximately twice as long as the write requests. This is expected since the read accesses require a write request to the FPGA and then a read request from the FPGA. This pattern between the read and write function calls is a check towards validating our results. Furthermore, we suspect the values to be accurate since the test system did not use data or instruction caching. Additionally,
for these tests, the prototype system was running at 59MHz. At this clock speed, the performance cost of 4 to 8 microseconds is equivalent to 200 to 400 assembly instructions. Given the number of C instructions for the middleware implementation, the number of assembly instructions is not surprising. For these reasons, we believe these are accurate measurements of the performance cost per middleware function call.

Table 3.3 gives a very insightful listing of the performance results. In the two low throughput tests, the cost due to the middleware layer is negligible. In both cases, the cost of the middleware layer would be small compared to network performance. However, in the high throughput tests, the middleware layer does incur a noticeable cost. This is especially true with the ftp download and http request cases. However, the overall costs are less than the execution time on the Pentium-90. Additionally, we think these are upper bounds on what the performance costs can be. First, DMA transfers are not considered. The function call counters count only byte, word, or doubleword accesses. DMA transfers are not considered since we could not get a cost measurement of each DMA access on the hostcard. Thus, the number of function calls are artificially high in the high throughput tests. We suspect the real middleware cost to be substantially lower with DMA support. Secondly, the cost of each middleware function call is high. These costs are calculated with the StrongARM clocked at 59MHz. If the StrongARM is clocked at 209MHz, the cost of each middleware layer function call would go down to approximately one fourth of the current value. However, it would be difficult to compare the execution time of the benchmarks on a 209MHz StrongARM versus on a 90MHz Pentium. Based on the Dryhstone benchmarks, the SA-1100 at 209 MHz has comparable VAXMIPS ratings to a Pentium-120 system. Thus, we suspect a Pentium-90 would have much better performance than a StrongARM at 59MHz. Overall, the results in Table 3.3 should be viewed as an upper bound on the performance cost of the middleware layer.

The performance tests show that the middleware layer incurs an acceptable level of performance costs. These performance costs would be much lower with full DMA support. Even so, the performance cost, when weighed against the benefits of the
middleware layer, shows that this is the right software approach for backplane communication.
Chapter 4

Related Work

This chapter focuses on some projects which are similar to the approach of MASC. Additionally, this chapter mentions some software projects in the industry which aid home automation and can be useful for MASC. Specifically, this chapter focuses the VMEbus and Jini. VMEbus is a bus based computation structure similar to our backplane and hostcard design. Jini is a set of network automation tools that simplifies building IA networks.

4.1 VMEbus

The concept of having modular computation resources is not new. It existed as early as 1980 with the VMEbus architecture. In VMEbus systems, multiple VME boards can be plugged into a single backplane. The boards include a variety of processor boards, I/O boards, memory boards, and peripheral boards. These boards can be plugged into the VME backplane which supports up to 64-bit wide address and data bus. Additionally, multiple computer boards can be used on the same VMEbus as a multiprocessor machine. The major differences between the MASC infrastructure and that of VMEbus are that:

- VMEbus requires a system controller to exist on the first board for arbitration reasons. On the MASC infrastructure, arbitration is done by the logic on the
backplane. Thus, the VMEbus requires a logic card with bus arbitration circuit to work. This restricts the capabilities that can be introduced to the system.

- The VMEbus is not intended to be a plug-and-play system. Thus, the software and hardware infrastructure does not support automatic reconfiguration of the system. Additionally, hot swapping of VME boards without power cycling is not possible.

- VMEbus does not have the performance of the MASC Cardbus standard. Although it has a 64 bit wide bus, its theoretical maximum throughput is only 80 MB/sec.

- The VMEbus infrastructure has a form-factor that is not easy for end users to reconfigure. The boards are fully exposed and are not enclosed in compact dimensions. Thus, they are cumbersome for the user to manipulate and use.

- VMEbus has several different form-factors which makes it more complicated to both end users and developers.

Thus, although the VMEbus has existed for nearly two decades, it does not have the performance, usability, and feature set of the MASC infrastructure. It can never be used in the ways we envisioned with the MASC information appliance infrastructure.

4.2 Jini

Recently, there has been widespread software development in home networking. Specifically, these software development efforts have aimed at automating network connections and setup. One well-publicized development project, the Jini project by Sun Microsystems Inc, is a set of network automation tools that operate on devices that can run Java programs. Jini handles the tasks of finding network connections and broadcasting different devices’ abilities. With Jini, any device that is plugged into the home network will be automatically recognized and used by other devices.
<table>
<thead>
<tr>
<th>Project</th>
<th>Company</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jini</td>
<td>Sun</td>
<td>Lets any Jini-enabled device or software module share services for &quot;spontaneous networking&quot; with other Jini devices.</td>
</tr>
<tr>
<td>Universal Plug and Play</td>
<td>Microsoft</td>
<td>Extends hardware recognition beyond PCs to let electronic devices easily connect without needing a computer.</td>
</tr>
<tr>
<td>Millennium</td>
<td>Microsoft</td>
<td>Allow group of computers to automatically divide up computing tasks across network.</td>
</tr>
<tr>
<td>T Spaces</td>
<td>IBM</td>
<td>Java-based system that allows computers, palm computers, or other devices to share messages, database queries, print jobs, or other network services.</td>
</tr>
<tr>
<td>Inferno</td>
<td>Lucent</td>
<td>Network operating system to let smart phones, set-top boxes, and other equipment plug into the network and run Java or other programs.</td>
</tr>
<tr>
<td>JetSend</td>
<td>HP</td>
<td>Communication technology that lets networked devices such as printers, scanners, or digital cameras negotiate common file formats for data exchange.</td>
</tr>
</tbody>
</table>

Table 4.1: **Industry software developments in automated networking**

on the network. This automated configuration of the network and devices simplifies application and device development. It also reduces the workload on the user. Sun believes that Jini will be useful for computing devices, cell phones, air conditioning systems, appliances, and other devices attached to the network. This level of software automation is essential for the consumer market since consumers will not accept complex devices difficult to setup or prone to error. As seen in table 4.1, other companies, such as Microsoft, Hewlett-Packard, IBM and Lucent are all working on similar technology [9]. All of the projects are aimed to create "spontaneous networking" of devices so that information appliance configuration would be automated. Although Jini is restricted to Java enabled devices, our infrastructure can still use it by enabling Java on the hostcard. Enabling Jini on our hostcard would enable a powerful software tool set automating the complex task of setting up a home network. It would push the MASC infrastructure much closer towards a complete product.
Chapter 5

Conclusion

This chapter discusses some of the lessons learned in building the MASC information appliance prototype, as well as what we accomplished with the prototype. Furthermore, we describe a number of possible improvements to the MASC infrastructure.

5.1 Evaluation of Goals

The original goals of this Masters project were to build a working prototype of the MASC infrastructure, to implement its software infrastructure, and to demonstrate its functionalities with software applications and performance benchmarks. Although not all of the goals were reached, the most difficult problems have been resolved. The prototype hardware was built with sufficient computational resources to function as a development platform. The backplane prototype illustrated the feasibility of an extremely cheap, reliable, and high performance interconnect. The hostcard was implemented using a high performance, low power, RISC CPU and a complete memory subsystem. Additionally, the I/O subsystem was fully implemented with flexibility and development in mind. The prototype hardware offered a glimpse at the potentials of reconfigurable computing and powerful information appliances, especially through applications such as the digital picture frame. Furthermore, the hardware prototype, by design, served as a development platform. The prototype hardware has been fully debugged and works well with the implemented software infrastructure.
On the software side, all of the system software layers have been implemented and debugged on the hardware infrastructure. The Angel debugger and boot-loader offers a quick and simple method to develop low-level applications. Furthermore, the Linux operating system has also been ported to the hostcard. A full development suite has been built for cross-development on PC Linux system. By using a commercial operating system such as Linux, we can port any number of Linux application to our infrastructure. Furthermore, we have implemented the MASC middleware layer, which allows existing device drivers to work with the hostcard I/O subsystem. By providing memory and I/O data access functions consistent with standard Linux kernel 2.2, nearly all existing Linux code can be used without significant modifications. All the system software except for the plug-and-play services have been implemented on the hostcard prototype. These software modules form a stable foundation upon which applications software can be developed.

Throughout this project, there were numerous engineering lessons about design tradeoffs, complexity, debugging, and solving obscure problems. These engineering lessons are essential in any large-scale project and proved invaluable in this one as well. Because this project involved both hardware design as well as software development, a wide variety of engineering ideas were used. The hardware implementation phase of the project utilized many prototyping techniques such as multiple prototype copies for backup purposes, extra signal visibility points, and extra space for component replacements. Common software practices such as abstraction layers, code modularity, and regression testing were repeatedly used throughout this project. All these applied techniques reinforced the lessons that I have learned throughout my undergraduate education at MIT.

5.2 Further Work

Despite the success of the MASC prototype, there are some remaining limitations. The hardware and software infrastructure has limited capabilities, is prone to user-induced failures, and is not user friendly. Significant work needs to be done to develop
the MASC prototype into the end product required in the next generation of information appliances. These information appliances would be integrated with the home network and revolutionize the way we think of appliances.

5.2.1 Hardware Improvements

There are a number of possible hardware improvements which can be made to the MASC prototype. They mostly involve replacing the development hardware with more efficient, smaller, and more cost-effective hardware components.

- The next version of the backplane will be much smaller when the XC4013 FPGA controller is replaced with a custom ASIC. In the current version, most of the footprint is taken up by the FPGA. On the embeddable backplane, by replacing the FPGA with a smaller ASIC, the backplane will have a footprint one quarter of the current size. The backplane size will then be constrained by the size and number of socket connectors. The next version of the standalone backplane will have more sockets and improved performance through the use of better controllers or a pipelined data bus. Another possible improvement would be an expandable backplane where additional sockets can be attached to the backplane through special connectors. This allows the user to expand the capabilities of an appliance without having to replace the embedded backplane.

- The next version of the hostcard should fit into the form-factor of a PCMCIA card. The prototype currently has a form-factor that is twice the area of a PCMCIA card. However, a large amount of this space is for debugging purposes. Additionally, the I/O subsystem with the FPGA and the dual-port RAMs can be replaced by an ASIC. The logic count of the FPGA is only about 40K and the dual port RAM supports two I/O buffers of 8KB capacity each. This is an acceptable amount of logic to be placed in an ASIC chip. Such a chip also means reduced latency and improved throughput. All of the components on the front half of the hostcard can be packed into this ASIC. Furthermore, replacing the ROM sockets with a thinner, smaller socket also creates more space on the
hostcard. With these hardware improvements, it would be easy to fit all these components within a PCMCIA-sized card.

5.2.2 Software Improvements

There are a number of software projects to improve the MASC software infrastructure. We want to make software development for the MASC infrastructure as simple as possible. To make this a reality, the following areas should be revised.

- First, a smaller and more efficient Linux kernel is essential. By removing the non-essential portions of the kernel, we can reduce the kernel space and free precious memory space for other programs. Additionally, a smaller kernel reduces the chance of system bugs and conflicts. Ideally, a micro-kernel based operating system can be adopted to replace the Linux monolithic kernel.

- Although the MASC middleware layer creates an abstraction layer separating the hostcard hardware from the device drivers, it is guaranteed to work with all Linux-supported PCMCIA devices. First, the middleware layer should be tested with all existing PCMCIA devices to ensure that they work properly. In certain cases where the device driver is not implemented properly, especially ones with poor abstraction implementation, the device drivers need to be modified to work with the middleware layer. Another project involves improving the performance of the middleware layer. To take advantage of the FIFO buffers on the hostcard, the middleware layer should context switch to pending threads when the executing threads is waiting for a backplane device to respond. This way, the CPU will waste the minimum number of cycles waiting for a device to respond.

- Multiple hostcard task management and support would be a very useful software improvement. Kernel-level support should be developed for recognizing when multiple hostcards are connected to the same backplane. Then a “master” hostcard will receive all of the tasks and pass them to the “slave” hostcards
on the backplane. The “master” hostcard will have to deal with load management, task priorities, and system resource allocations. One way in which the job of the “master” hostcard can be simplified is by reducing the complexity of the parallel systems. By dealing with parallelism at the higher granularity of independent tasks, problems such as data consistency, synchronization, and deadlocks can be avoided. Overall, this improvement is essential to allow scalability of computation and to boost performance in a multi-tasking environment.

- Additional software improvements would be within applications that can take full advantage of the reconfigurable computational resources of the MASC infrastructure. These applications could include behavioral artificial intelligence programs and information processing applications. These applications could introduce new functionalities and features into the appliance, as well as expose existing ones to other appliances on the network. Development with other operating systems as well as network automation software such as Jini would help make the MASC infrastructure more user-friendly and easier to use.

With these improvements, it would not be too difficult to envision a truly user-friendly, modular, reconfigurable computation infrastructure based on MASC. One in which many appliances, tools, and electronics contain interfaces for computation. An user can simply plug in one of a multitude of computer-on-a-cards. Instantaneously, these home electronics become “intelligent”. By adding network cards, I/O cards, and any other peripheral devices, the user can construct a computer fitting the task at hand. With so much reconfigurable computation and communication power local to each piece of electronics, powerful software can take full advantage of the electronic appliance. In a home, some possible uses of the MASC IA infrastructures are:

- When a person misses his favorite television show, the television information appliance would recognize that the TV is not on and tell the recording device (VCR, personal computer, or standalone storage device) to power on and request that the show be recorded.
• If the phone rings and the television is turned on, then the information appliance would reduce the volume of the television and notify the user with caller identification information on the screen of the television.

• If the parent at work wants to keep an eye on the children at home, he or she can retrieve images from the information appliance camera at home, which is connected to the network through network cards.

• The water heater, air conditioner, and heater in the house are linked to information appliances which gather information about the activity of the user. If the user is out of the house, then these units shut down to save power.

• The electricity meter can download information about rates from different electricity companies and, depending on the usage pattern, find the cheapest rate for the user.

AI programs which make intelligent suggestions, information retrieval systems, and data processing are only some of the possible software which can now run on the appliances. For each of these possible applications, there are literally hundreds more which have not been thought about. With computation available in such a modular, configurable, user-friendly package, there would be innumerable ways in which the MASC infrastructure can be applied.

The methods in which computation can then be applied will change our perception of what a computer is. It might change our whole perception of what computation is. It might change how we perceive and receive information in our daily lives. Finally, it might change the way we look at the world.
Appendix A

Prototype Hardware

A.1 MASC Prototype Backplane

Since it is fixed to the appliance, the backplane must be cheap, simple, small, and reliable. It should support communication among multiple computer hostcards and peripheral cards with low latency and high throughput. It must also be robust and compatible with multiple generations of hostcards and peripheral cards. Adopting a well-known industry standard for the backplane has its advantages. It allows the MASC infrastructure to leverage existing technology on the market. Using a tested, documented, and widely-used industry standard not only cuts down on development time, but also allows our infrastructure to instantaneously use all existing products based on that standard.

Based on these requirements, we chose the backplane to follow the specifications of the Cardbus PC-Card standard defined by PCMCIA. The Cardbus standard combines the low-power, small form-factor, and plug-and-play capability of the PCMCIA standard with the high performance of PCI system bus [3]. The Cardbus standard uses 68-pin connectors and the form-factor of the PCMCIA 16-bit PC card. However, it supports a 32-bit Combined Address Data (CAD) bus with a theoretical throughput of 132 MB/second. This is the same theoretical performance as the PCI system bus. By choosing this standard, we utilize a clean, well-designed, and universally defined standard. Most backplane communication problems, such as arbitration, signal
Figure A-1: Backplane layout

integrity, and control logic, have been solved in the Cardbus standard. Finally, using the Cardbus standard allows the MASC infrastructure to utilize the wide array of existing Cardbus products. The products include network cards, memory cards, hard drives, modems, camera cards, IEEE-1394 firewire connectors, and multi-function cards.

However, there are limitations with choosing the Cardbus standard for the MASC backplane infrastructure. Most notably, this design choice puts a theoretical limit on the maximum performance of the backplane. At least for the prototype system, the 132 MB/sec throughput is more than sufficient. An additional problem is the Cardbus standard’s 68-pin card-socket connector. Of these 68-pins, 65 are used either by the 32-bit CAD bus or logic control signal. There is not much room for growth if the MASC infrastructure needs to add signals. For example, the hostcard might need to add more data lines through the backplane. The low-pin count restricts how many signal lines can be sent through the backplane.

Although we built multiple versions of the prototype backplane, they all use similar
components and have similar design layouts. The Cardbus standard has well-defined requirements on the properties of the backplane bus. As seen in diagram A-1, the main data bus is the 32-bit Common Address and Data (CAD) bus which connects to all the sockets. On the prototype backplane, we used the Xilinx XC4013 FPGA to implement the control logic. By using an FPGA, we get the flexibility of programmable logic and the ability to add new features into the backplane.

A.1.1 Standalone Backplane

The standalone prototype backplane has six sockets. The main components of this backplane are the Xilinx XC4013 FPGA, the 3 pairs of 68-pin FPC Cardbus connectors, and a clock circuit (figure A-2). We mainly used this prototype backplane to test proof of concept and for debugging purposes. It was intended for use with multiple hostcards and multiple peripheral devices.

A.1.2 Embeddable Backplane

In addition to the full-sized backplane, we also implemented a smaller backplane
Figure A-3: Embeddable backplane

designed to be embedded in appliances. This embeddable backplane emphasizes its design in simplicity and minimization of size (figure A-3). The resulting backplane is very compact, and cheap. This prototype backplane has only two sockets, one for the computer card and one for the network card. Its dimensions are only 4.4 cm by 5.7 cm. Again, the only components on the backplane were the Cardbus FPC socket connectors, the XC4013 FPGA, and the clock circuit.

A.2 MASC Prototype Hostcard

We want the MASC hostcard to use low-power, to have high-performance, and to be compatible with the Cardbus standard of the backplane. Our hostcard prototype satisfies all of these requirements with carefully chosen components and a well-designed system architecture.

As seen in figure A-4, the hostcard architecture contains the Intel SA-1100 CPU, a memory subsystem, and a backplane I/O subsystem. The key component for the hostcard is the CPU. The CPU of the prototype hostcard is the Intel SA-1100 microprocessor. It is a member of the StrongARM microprocessor family. We chose SA-1100, because it delivers a combination of high performance (230 Dhrystone VAXMIPS @ 200 MHz), low power (250mW @ 1.5V) and extensive peripheral support for a wide
Figure A-4: Hostcard layout
Figure A-5: **StrongARM-1100 CPU**

range of applications [2].

Another main reason we chose the SA-1100 is the integration of many features on the chip. As seen in figure A-5 [2], the SA-1100 is very much a system on a chip. It has a wide range of operating speeds and supports both an instruction and a data cache. The Memory and PCMCIA Control Module (MPCM), System Control Module (SCM), and Peripheral Control Module (PCM) are all tightly integrated with the CPU core. The Memory and PCMCIA Control Module (MPCM) can interface with up to 4 banks or 512 MB of DRAM, 4 banks of programmable memory, and 2 PCMCIA slots. The System Control Module (SCM) interfaces through 28 general-
purpose I/O (GPIO) pins for interrupts, clock signals, and specialized I/O. It also contains the power management and reset controllers. The Peripheral Control Module (PCM) has pins for a 16-bit color LCD controller, one infrared port, two serial ports, a multimedia communication port, and an USB port. Additionally, it has a clean pin allocation scheme that simplifies building a computer system around the SA-1100. In fact, no external logic is required to turn the SA-1100 into a complete computer system.

A.2.1 Memory Subsystem

With the SA-1100, up to 4 banks of DRAM and 4 banks of ROM could be addressed. For the hostcard prototype, the memory subsystem has two banks of 16 MB DRAM and one bank of 4MB Flash ROM. Although more memory could be added to the hostcard prototype, size and complexity were weighed against the additional storage space. The SA-1100 connects to the memory subsystem via its Memory and PCMCIA Control Module (MPCM). The MPCM interfaces the memory banks with a 32-bit data bus, 26-bit address bus, 4 CAS, 4 RAS and 4 CS lines.

For DRAM, the hostcard has four Micron Technology MT4LC4M16R6TG-5 64 Megabit synchronous DRAM chips. The 4Mx16 DRAM chips are arranged in two DRAM banks with a 32-bit wide data path for each DRAM bank. Some of the reasons that we chose the Micron DRAM chips were its availability, price, and performance. Specifically, it has a 50 nanoseconds access time and is available in the 50-pin TSOP package. This package has one of the smallest footprints at dimensions of 10.2 cm by 21 cm by 1.2 cm. The small footprint in conjunction with the thinness of the chip reduces the difficulty of fitting all the components onto the backplane.

The hostcard also has two AMD AM29LV160BB 16 megabit flash memory chips. We chose the AMD chips because they are easy to reprogram, have significant storage space, and have high performance (80 nanoseconds access time). Because the prototype is also a software development system, 4 MB of ROM space is deemed the minimum for development flexibility. Since a prototype software system contains a boot-loader (less than 128KB), a Linux kernel (512KB-1MB), and a compressed
image disk (1.5 MB-2MB), 4MB is the minimum ROM space required to fit all the storage data.

**A.2.2 I/O Subsystem**

The I/O subsystem serves as the bridge between the system bus and the backplane. The I/O subsystem consists of one Xilinx XC4062 FPGA, 3 IDT 4kx16 dual-port static RAM, and 2 Atmel 512 kilobit EEPROM. On the system bus side, the FPGA is connected as DRAM bank 3 to both the 32-bit data bus and the 26-bit address bus. Thus, read and write requests for DRAM bank 3 are handled exclusively by the FPGA logic. On the Cardbus side, the FPGA is connected to the 32 bit CAD bus and other Cardbus control signals.

In addition to the aforementioned subsystems, we added several ports to give the prototype hostcard more flexibility for debugging and demonstration purposes.

- **LCD Port** - we added a 8-bit LCD port connected directly to the SA-1100 LCD controller. The port has the following signal lines: 8 LCD data lines, frame clock, pixel clock, line clock, and AC bias line. These signal lines are standard LCD control lines and can drive a wide variety of LCD panels.

- **Serial port** - the serial port is routed through a RS-232 circuit to the UART3 port
of the SA-1100. The serial port is used for debugging purposes. The serial port allows direct communication to the SA-1100 without going through network devices and the backplane. It also is a communication channel for the Angel Debug Monitor, a download channel for the Linux kernel, and a connection method for the serial console.

- Parallel port- the parallel port serves as additional programmable I/O logic pins. They are connected to pins on the Xilinx FPGA. They can be used for a variety of digital signal generations via the FPGA logic.

As seen in figure A-6, all of the components fit nicely on our prototype hostcard in an area of less than 15 cm by 7 cm. The hostcard used a 6 layer printed circuit board with 6 mil traces. The hostcard layout was designed using the Accel Technologies Tango PCB design suite. We also used the Spectra autorouter to route all signal lines on the board.
Bibliography


