Linux PCI Shared Memory Device Drivers for the Cal Poly Intelligent Network Interface Card

A Senior Project Report Presented to the Computer Science Department

By

Mark McClelland

California Polytechnic State University, San Luis Obispo

Date Submitted: June 18, 2001
Advisor: Dr. Hugh Smith
ABSTRACT

The Cal Poly Intelligent Network Interface Card (CiNIC) Project is a research group at California Polytechnic State University, San Luis Obispo, that is funded by the 3Com Corporation. The goals of the group are to enhance networking performance and capabilities, in order to enable a faster and more powerful Internet.

This paper describes a pair of device drivers that facilitate sharing of memory between a Linux-based intelligent network interface card (NIC) and its Linux host system. The objective is to support the offloading of TCP/IP stack processing from the host system to the NIC. This is done by passing the unprocessed socket call parameters and data structures between the host system and the NIC, by means of the shared memory device drivers. The process of implementing the drivers is discussed, along with the details of their internal operation. Also included in this paper is a discussion of the specific constraints imposed upon such device drivers by our particular hardware platform, and by the Linux operating system.

Test procedures are also described, and real-world results are included that show that the drivers are working as designed.
ACKNOWLEDGMENTS

First, I would like to thank Dr. Hugh Smith for being my advisor on this senior project. His continuing support for the CiNIC Project, and for my work, is what made this senior project possible. Also, his excellent networking courses provided me with invaluable experience that came in handy countless times throughout this project.

I would also like to thank Dr. James Harris for his constant vigilance in keeping the project running smoothly, and for securing the resources and support from the 3Com Corporation that have made this project a reality. Of course, many thanks go to 3Com for graciously providing their funding and time.

Finally, I would like to thank the student members of the project for their hard work. Special thanks go to Jim Fischer, who taught me a great deal about the CiNIC platform, and to Rob McCready, Jason Hatashita, and Bo Wu, for the code they contributed to the project.

Table of Contents

List of Tables ............................................................................................................... vi
List of Figures ............................................................................................................. vii
1. Introduction...............................................................................................................1
2. Background .............................................................................................................2
  2.1. Overall System Design.................................................................................. 2
  2.2. Linux .............................................................................................................. 3
  2.3. PCI.................................................................................................................. 4
  2.4. 21285............................................................................................................. 8
List of Tables
Table I: Standard PCI Configuration Space .............................................................. 6
Table II: 21554 Primary Interface Base Address Registers ..................................... 13
Table III: 21554 Secondary Interface Base Address Registers ............................. 13
Table IV: Files Modified by Big Physical Area Patch ........................................... 21
Table V: Contents of the 21554 Downstream Memory 1 Setup Register ............... 43

List of Figures
Figure 1: Schematic of CiNIC Hardware ............................................................... 2
Figure 2: Downstream Address Translation Example ......................................... 12
Figure 3: PCI Address Translation Example ..................................................... 14
Figure 4: Modules in the Second Implementation ............................................. 38
Figure 5: Co-Host Memory Map ................................................................. 41
1. Introduction

The main goals of the Cal Poly Intelligent Network Interface Card CiNIC Project are:

1. To enhance network performance and system responsiveness by offloading the TCP/IP stack from a host system onto a co-processor.

2. To implement additional features such as firewalling and IPv6 on the co-processor, so that these features can be used even if the host system does not directly support them.

To facilitate data transfer between the host and the co-processor, I developed a pair of device drivers that set up a shared memory communications channel between the host system and the co-processor. This document describes the development and testing of these device drivers. Background information on the CiNIC platform that is necessary for a complete understanding of the device drivers is provided as well.
2. Background

This chapter provides a basic description of the hardware and software components that underlie the shared memory device drivers.

2.1. Overall System Design

The CiNIC hardware test platform consists of the components shown in Figure 1:

The host computer system is a standard Intel x86-based Personal Computer running the Linux operating system. The 21554 is a PCI-to-PCI bridge device that resides on a PCI backplane. This backplane plugs into host system's PCI bus, which is referred to in this document as the "primary PCI bus."
The 21554 backplane provides four PCI sockets that are connected to a second PCI bus, referred to as the "secondary PCI bus," which is on the opposite side of the bridge from the host. The EBSA-285 co-processor board is plugged into one of these secondary PCI bus sockets. The other sockets are used for a network interface card, an EIDE controller, and a PCI bus analyzer riser card.

The EBSA-285 co-processor, referred to as the "EBSA" or the "co-host" throughout this document, is an embedded computer board that is based on the StrongARM SA-110 microprocessor. The processor is connected to a 21285 core-logic chipset, which provides interfaces to SDRAM, PCI, and other co-processor components.

### 2.2. Linux

Linux is a free and open-source operating system kernel that has been ported to many processor types and hardware platforms. The primary motivation for using Linux in the CiNIC project is that the source code is publicly available and freely modifiable, as long as any changes we make are released back into the public domain. This allows us to customize our platform with new features and bug fixes as necessary.

The host system runs an Intel x86 version of the 2.4.x-series Linux kernel, which is part of the RedHat Linux 7.1 operating system that is installed on the host. The
EBSA-285 runs the StrongARM port of the 2.4.x-series kernel. RedHat Linux is not available for this architecture, so we installed our own custom operating environment piece-by-piece.

2.3. PCI

The PCI (Peripheral Component Interconnect) bus is a 32-bit\(^1\) parallel data bus that is used in most personal computers manufactured today. Data transfers on the bus are known as transactions. Up to 256 busses may be connected together using PCI bridges [9].

There are two different types of bridges:

**Transparent**: These replicate all transactions, with a few exceptions, from each attached bus to all other attached busses, so that every bus ends up having the same transactions.

**Non-transparent**: These can be configured to replicate specific transactions, and block all others from passing through the bridge. The decision to pass the transaction can be based on its type, its address, or a multitude of other reasons [5].

Up to 32 physical devices may be attached to each PCI bus, and each device may

---

\(^1\)PCI has a 64-bit data bus extension that is not used in our current CiNIC implementation.
have up to 8 functions. Functions provide configuration interfaces and data interfaces that are logically separate from the interfaces of other functions on the device. Each function in a PCI system can be addressed using a combination of its bus, device, and function number.

The device that is sending a PCI transaction is called the *initiator*, and the devices that receive it are called the *targets*. Some transactions are *globally addressed*, meaning that they are sent to all devices, and are identified by a 32 or 64 bit address. Other transactions are *geographically addressed* (e.g. configuration transactions), meaning that they are directed to a specific address on a specific function [9].

PCI transactions fall into the following three categories:

**I/O Transactions:** These are used to read and write data serially, usually to an input/output hardware interface. When a target receives an I/O transaction, it performs an action automatically, such as sending the transaction's data payload to an RS-232 port. Up to 4 GB of I/O ports may be globally addressed.

**Memory Transactions:** These perform random access reads and writes on a device's storage space. The storage is usually a block of registers, a ROM, or RAM. Up to $2^{64}$ bytes of memory space may be globally addressed [7].

**Configuration Transactions:** These are used to configure a specific PCI
function. Each function in a PCI device has 256 bytes of geographically
addressed configuration space. Type 1 Configuration Transactions are used
to configure transparent bridges, and Type 0 Configuration Transactions
are used to configure non-transparent bridges and all other devices [10].

Table I: Standard PCI Configuration Space [9]

<table>
<thead>
<tr>
<th>Byte 3</th>
<th>Byte 2</th>
<th>Byte 1</th>
<th>Byte 0</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device ID</td>
<td>Vendor ID</td>
<td>00h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status Register</td>
<td>Command Register</td>
<td>04h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class (Bytes 2,3)</td>
<td>Class (byte1)</td>
<td>08h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIST</td>
<td>Header Type</td>
<td>0Ch</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Latency Timer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cache Line</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Address Register 0</td>
<td></td>
<td>10h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Address Register 1</td>
<td></td>
<td>14h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Address Register 2</td>
<td></td>
<td>18h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Address Register 3</td>
<td></td>
<td>1Ch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Address Register 4</td>
<td></td>
<td>20h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base Address Register 5</td>
<td></td>
<td>24h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CardBus CIS Pointer</td>
<td></td>
<td>28h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subsystem ID</td>
<td>Subsystem Vendor ID</td>
<td>2Ch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expansion ROM Address</td>
<td></td>
<td>30h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>34h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>38h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max Latency</td>
<td>Min_Gnt</td>
<td>IRQ Pin</td>
<td>IRQ Line</td>
<td>3Ch</td>
</tr>
</tbody>
</table>

Transactions are normally initiated by the PCI host controller, which acts as a bridge
to the PCI bus and the system's processor. This controller is usually the bus
arbiter too, which means that it has the authority to allow other devices become
initiator. The process of a device becoming the initiator is called bus-mastering [10].

PCI also supports interrupts, which allow a device to notify another device about
some specific event or condition. Each device has four interrupt pins, and each
function may use one of these pins. Interrupt pins may be shared between functions,
or even between devices.

To make it easier for users to add devices to their systems, PCI has a feature known as Plug and Play. Plug and Play allows the hardware configuration of devices to be set up automatically by software running on the host processor. Each PCI function has a vendor ID for identifying its manufacturer, a device ID for identifying a manufacturer's specific product, and a class code for identifying functions with manufacturer-independent programming interfaces. These allow the system software to identify what device driver a function needs, without any intervention from the user.

All PCI functions have six Base Address Registers, or BARs, in their configuration space. These are 32-bit registers that hold the start of memory and I/O address ranges that belong to the function. These control programmable address decoders in the function's hardware, and they can be modified by the system software when necessary to avoid address conflicts [9].

When a device is powered on or reset, all of its BARs are set to zero [10]. Each BAR must be configured using the following procedure:

1. Use a Type 0 Configuration Write transaction to write 0xFFFFFFFF to the BAR. This causes the device to store a configuration bit mask in the BAR.
2. Use a Type 0 Configuration Read transaction to read the BAR's configuration bit mask. See Table V in Chapter 6 for a detailed description of the bit mask. If the bit mask is all zero, this means that the BAR is permanently disabled.

3. Determine the required size of the address space from the bit mask, and find a suitable range of contiguous PCI addresses that is not being used.

4. Write the starting address of the range to the BAR with a Type 0 Configuration Write transaction [9].

2.4. 21285

The Intel 21285 is the core logic chip of the EBSA-285. Its primary function is to interface the SA-110 microprocessor to the rest of the system [8]. It has the following features that are relevant to the CiNIC project:

- **SDRAM interface**: Provides address bus and timing control
- **Flash ROM interface**: Allows itself to be initialized with settings stored in a flash ROM
- **PCI bus interface**: Provides access between SA-110 and a 32-bit PCI bus, and acts as bus arbiter
- **SDRAM window**: Provides transparent access to SDRAM via the PCI bus
- **Miscellaneous controllers**: One UART (serial port), DMA controllers, interrupt controller

These features are configured with the 21285's Control and Status Registers, or
"CSRs". These include the standard PCI Configuration Space Registers, 21285-specific PCI Control and Status Registers, and SA-110 Control and Status Registers. They can be accessed or modified by the SA-110 at memory locations 42000000h to 4200036Ch. Additionally, most of them are accessible via the PCI bus within the 21285's PCI configuration space.

Most important to the CiNIC project are the CSRs at offsets 100h and 104h. These are the SDRAM Base Address Mask Register and the SDRAM Base Address Offset Register, respectively. These control the mapping of the SDRAM window onto the PCI bus. This allows the host to read and write the shared memory stored on the EBSA-285 co-host, across the primary and secondary PCI busses and through the 21554 bridge.

It is important to note that reads or writes between the SA-110 and the shared memory never utilize PCI transactions, and never pass through the SDRAM window. This is because it is most efficient to simply access the memory directly. Conversely, access to the shared memory through the SDRAM window from the host always requires PCI transactions, since the co-host stores the "definitive copy of the memory". The host has no automatic way to know if any local copies of the shared memory are still the same as the "definitive copy" on the co-host. However, it is always possible for a specific data transfer protocol to define when changes to the
shared memory are valid, allowing copies to be cached on the host.

2.5. 21554

The Intel 21554 chip is a non-transparent PCI-to-PCI bridge [7]. Non-transparent means that each "side" of the bridge is an independent bus, and the bridge looks just like a normal PCI device to the systems on each side. All communication between the busses is accomplished by programming the bridge to pass through I/O and memory transactions that fall within a specified range of addresses. Other PCI functions, such as interrupts, are passed to the bus on the opposite side of the bridge by setting special flag registers in the bridge's configuration space.

The 21554 also acts as an address translator [7]. The address spaces on each side of the 21554 are completely independent. The bridge can be set up to pass transactions in a given address range on one side to a different address range on the other side, as long as the ranges are the same size. This lets the host system on each side take full advantage of PCI Plug and Play capabilities, by allowing an address range to be relocated in case of a conflict, without affecting the other side of the bridge.

Like the 21285, the 21554 also has CSRs. Since the 21554 is not attached to the system bus, they cannot be set using standard I/O instructions or memory reads and writes [7]. They must be set using Type 0 PCI configuration transactions, which can
be done in the Linux kernel with functions such as `pci_write_config_word()` [4].

The 21554 has two separate PCI bus interfaces, referred to as the "primary" and "secondary" interfaces. In our particular setup, the host is attached to the primary interface, and the EBSA-285 is attached to the secondary interface. Both sides are fairly similar in terms of their operating capabilities. However, the system on the secondary interface has more configuration capabilities. This puts the burden of configuration on the EBSA-285, but it also means that the host configuration can be kept very simple. Like the 21285, the 21554 can also optionally read its initial configuration from a serial ROM [7].

Transactions initiated on the primary bus that are passed to the secondary bus are called "downstream" transactions, and transactions initiated on the secondary bus that are passed to the primary bus are called "upstream" transactions [7]. Our implementation only uses the downstream functionality of the 21554, which allows the host to initiate transactions that are passed to the co-host. Upstream transactions are not used because the co-host already has the definitive copy of the shared memory stored locally.
The base addresses of the memory ranges on the *initiating* side of the transactions (Base 1 in Figure 2) are set using the standard PCI Plug and Play procedure, which is
automatically performed by the Linux OS. These addresses are stored in the 21554’s PCI Base Address Registers (BARs). There are four downstream BARs on the primary interface, and three upstream BARs on the secondary interface [7].

There is a complete copy of the primary PCI configuration space on the secondary interface, and vice-versa. The secondary interface may read or write any register in either the primary or secondary configuration space. The primary interface may only access the secondary configuration space if the Primary Lockout configuration space bit has been cleared [7].

Table II: 21554 Primary Interface Base Address Registers [7]

<table>
<thead>
<tr>
<th>Base Address Register</th>
<th>Size (bytes)</th>
<th>Primary CSRs</th>
<th>Secondary CSRs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary CSR / Downstream Memory 0 BAR</td>
<td>4K to 2G</td>
<td>10h - 13h</td>
<td>50h - 53h</td>
</tr>
<tr>
<td>Primary CSR I/O BAR</td>
<td>256</td>
<td>14h - 17h</td>
<td>54h - 57h</td>
</tr>
<tr>
<td>Downstream I/O or Memory 1 BAR</td>
<td>64 to 256 (I/O)</td>
<td>18h - 21h</td>
<td>58h - 61h</td>
</tr>
<tr>
<td></td>
<td>4K to 2G (mem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Downstream Memory 2 BAR</td>
<td>4K to 2G</td>
<td>1Ch - 1Fh</td>
<td>5Ch - 5Fh</td>
</tr>
<tr>
<td>Downstream Memory 3 BAR (64-bit)</td>
<td>4K to 2^63</td>
<td>20h - 27h</td>
<td>60h - 67h</td>
</tr>
<tr>
<td>Primary Expansion ROM BAR</td>
<td>4K to 16M</td>
<td>30h - 33h</td>
<td>70h - 73h</td>
</tr>
</tbody>
</table>

Table III: 21554 Secondary Interface Base Address Registers [7]

<table>
<thead>
<tr>
<th>Base Address Register</th>
<th>Size (bytes)</th>
<th>Primary CSRs</th>
<th>Secondary CSRs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secondary CSR Memory BAR</td>
<td>4K</td>
<td>50h - 53h</td>
<td>10h - 13h</td>
</tr>
<tr>
<td>Secondary CSR I/O BAR</td>
<td>256</td>
<td>54h - 57h</td>
<td>14h - 17h</td>
</tr>
<tr>
<td>Upstream I/O or Memory 0 BAR</td>
<td>64 to 256 (I/O)</td>
<td>58h - 61h</td>
<td>18h - 21h</td>
</tr>
<tr>
<td></td>
<td>4K to 2G (mem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Upstream Memory 1 BAR</td>
<td>4K to 2G</td>
<td>5Ch - 5Fh</td>
<td>1Ch - 1Fh</td>
</tr>
<tr>
<td>Upstream Memory 2 BAR</td>
<td>16K to 256M</td>
<td>60h - 63h</td>
<td>20h - 23h</td>
</tr>
</tbody>
</table>
The base addresses of the memory ranges on receiving side of the transactions (Base 2 in Figure 2), as well as the sizes of the ranges, are generally set by the processor on the secondary side. They could also be programmed into the 21554's serial ROM, to ease configuration and prevent accidental modification. These parameters are written to the Device-Specific Configuration Address Map, which begins at CSR offset 80h on both interfaces.¹

¹See Figure 7-3 in [7] for more information on the these CSRs
3. Allocation of the Shared Memory

3.1. Technical Constraints

The PCI standard has the requirement that the bus address of an address region be naturally aligned. This means that the starting address of a PCI memory region must be at an address that is a multiple of the size of the region. This is because PCI devices generally implement a simple and fast address mapping scheme, that uses a base address and a mask value to determine whether an address falls within a particular region [10].

Figure 3 shows how the 21285 translates SDRAM addresses. In this example, a PCI address is converted to an SDRAM address. The BAR holds the starting PCI address of the window, and the SDRAM offset register holds the starting SDRAM address of the window[6]. The conversion procedure is as follows:

<table>
<thead>
<tr>
<th>PCI Address</th>
<th>BAR</th>
<th>Mask</th>
<th>SDRAM Offset</th>
<th>SDRAM Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 6 4 0 0 0 0 9</td>
<td>8 0 0 0 0 0 0 0</td>
<td>0 F F C 0 0 0 0</td>
<td>2 0 0 0 0 0 0 0</td>
<td>2 6 4 0 0 0 0 9</td>
</tr>
</tbody>
</table>
1. Copy bits 2 through 17 of the PCI address to the SDRAM address. Because PCI addresses are aligned on four-byte boundaries, bits 0 and 1 are always zero.

2. For bits 18 through 31, if a bit in the mask register is 0, copy the corresponding bit in the SDRAM offset register to the corresponding location in the SDRAM address.

3. For bits 18 through 31, if a bit in the mask register is 1, copy the corresponding bit from the PCI address to the corresponding location in the SDRAM address.

The most important detail to notice is that bits in the SDRAM offset register can only be 1 if the corresponding mask bits are zero, otherwise the SDRAM address will not be computed correctly. The address, after translation, will always be equal to a multiple of the window size plus the difference between the PCI address and the BAR. The resulting address will be the same even if the SDRAM offset register is a multiple of the window size, plus some additional offset. This additional offset will be lost during the translation. Therefore, the SDRAM address of the shared memory window that we will be exporting to the host via the PCI bus must be naturally aligned. Additionally, the shared memory must be physically contiguous since the ARM processor does not allow noncontiguous physical pages to be mapped into a contiguous block on the bus [6].

These were particularly difficult constraints to adhere to during the development of
the CiNIC, since the Linux kernel does not have any built-in mechanism for allocating large (> 128K) regions of physically contiguous memory [9]. Neither can it guarantee that the memory is aligned on a multiple of its size. The smallest SDRAM window size supported by the 21285 is 256K [6], so using a window of 128K or less is not possible. Additionally, there is no guarantee that there will be 128K of contiguous RAM available by the time my device driver allocates the memory, since the free memory may have been fragmented into small pieces by that time.

3.2. Implementation

When I began working on the driver, I had little experience working with the virtual memory (VM) subsystem of the Linux kernel. There are many ways to allocate memory under Linux, each with its own peculiarities and specific range of applications. Finding a function that could allocate a large amount of memory that is physically contiguous, with near 100% reliability, would not be easy even if it had existed in the stock Linux kernel.

My first plan was to use `kmalloc()`, or its underlying function, `get_free_pages()`, to allocate as much contiguous memory as possible at one time. `get_free_pages()` keeps track of free memory in groups of like-sized physically contiguous blocks, using a method called the "buddy system." Each block size is a power of `PAGE_SIZE`, which happens to be 4096 bytes on the StrongARM
platform. The "buddy system" groups like-sized blocks into separate linked lists for each size. Each list is ordered by the physical addresses of the blocks it contains. If a larger block is needed than what is available, then smaller contiguous blocks are merged into a block of the requested size. Likewise, if a smaller-sized allocation is requested than what is available, a larger block is broken into smaller blocks [1].

As it turns out, \texttt{kmalloc()} does not guarantee that the memory will begin or end on page boundaries, so alignment becomes an issue. It is also notorious for wasting space in certain cases, which is not a desirable attribute in an embedded environment [9].

However, \texttt{get\_free\_pages()} does guarantee complete contiguous pages. Accordingly, \texttt{kmalloc()} takes as its size argument a number from 0 to 5, representing \( \lg(N) \), where \( N \) is the number of pages being requested [9]. Therefore, possible sizes are 4K, 8K, 16K, 32K, 64K, and finally 128K, which unfortunately is still not enough to meet the 256K minimum for the SDRAM window. However, as long as the 128K blocks \texttt{get\_free\_pages()} returns are physically contiguous, they can be joined together into blocks that are large enough to meet the minimum requirements.

For a 256K window size, the worst possible case would be that the allocated memory begins at a multiple of 256K, plus one page. This would require that \( 512K - 4K = \)
508K be allocated to ensure proper natural alignment. This exact amount could be theoretically allocated by `get_free_pages()`, by allocating three 128K blocks, and one block of each remaining different size. A function would need to be written that allocates blocks of the proper sizes, ensures that they are contiguous, and if it fails, tries again with more blocks in the hope that a large enough contiguous memory space can be found. This is very resource-intensive since blocks allocated during the failed attempts cannot be freed until an attempt succeeds, otherwise `get_free_pages()` will most likely allocate the same blocks that were allocated in the previous attempt. Even then, severe memory fragmentation could render all attempts at securing the memory fruitless. This is especially likely in a system that has only 16 megabytes of RAM, like our EBSA-285 had when this problem was first addressed\(^1\).

These issues prompted me to find a way to allocate the memory at boot-time, when fragmentation has not yet occurred, and when more free memory is available. The kernel has a number of built-in mechanisms for allocating memory at boot-time, but they would require modifications to key source files in the kernel. There is also a trend within the Linux kernel developer community to move away from allocating memory at boot-time, since it can never be freed. We were working with a 2.3.x-series kernel at the time, so there was no guarantee that these methods, which have

\(^1\)It now has 128 megabytes of RAM.
already changed dramatically since the 2.2.x kernel series, would remain the same by
the release of the 2.4.0 kernel. I also didn't like the idea of having to use my own
custom version of the kernel, since every new kernel release would have to be re-
synchronized with my changes. Fortunately, there are other ways of reserving
memory early in the boot process.

One of these ways is to pass to the kernel the "mem=XM" parameter on its boot loader
command-line. Until recently, the kernel could not always find all available RAM in
the system on the i386 architecture, so this parameter was added to allow the RAM
size to be forced to whatever the user specifies [9]. Normally, this parameter is used
to increase the amount of RAM that the Linux kernel can see, but it can also be used
to decrease the available RAM. For example, if the kernel is passed the "mem=15M"
parameter on a system with 16 megabytes of primary RAM, the highest one megabyte
will be completely ignored by Linux.

Thus, the memory can be used without having to tell the operating system that you
"own" it and that don't want anything to write over it. This also has the added
advantage that the one megabyte block is automatically naturally aligned, and
therefore a shared memory size of one megabyte can be achieved with no loss of
additional space. However, since the operating system cannot see it, the kernel's page

---

1See linux/Documentation/kernel-parameters.txt in [4]
allocation and management code cannot deal with it. This means that if virtual address space mappings of the memory are desired (e.g. for access by kernel functions that expect kernel virtual addresses), the page tables must be manually set up by my driver. This is by no means a trivial task. Fortunately, there are kernel functions such as ioremap() and vremap() that can do this [9]. However, there is no guarantee that they will function correctly in this "no man's land" of system address space, with respect to cache coherency and other issues. There is also the minor disadvantage that memory can only be "allocated" with one megabyte precision using this technique, which could be a problem on systems with very limited RAM. Therefore, I continued to look for a better way to achieve the requirements of the shared memory allocation with a minimum of disadvantages and complexity.

I found an excellent compromise between a standard boot-time allocation patch and the "mem" kernel parameter. This was an unofficial kernel modification known as the "Big Physical Area" patch¹. This patch allows for a large region of physically contiguous memory to be reserved at boot time, and to be allocated from at any time by kernel code. The kernel source files changed or added by the patch are described in Table IV.

¹This patch is available on the source code CD-ROM described Appendix C.
Table IV: Files Modified by Big Physical Area Patch

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch/i386/config.in</td>
<td>Modified to provide a kernel configuration option for enabling the patch, when compiling for an Intel x86 CPU</td>
</tr>
<tr>
<td>Documentation/Configure.help</td>
<td>Modified to add help text for the kernel configuration option</td>
</tr>
<tr>
<td>Documentation/bigphysarea.txt</td>
<td>Documentation for the patch</td>
</tr>
<tr>
<td>kernel/ksyms.c</td>
<td>Modified to export allocation/deallocation functions to kernel modules</td>
</tr>
<tr>
<td>include/linux/bigphysarea.h</td>
<td>Prototypes for allocation/deallocation functions</td>
</tr>
<tr>
<td>mm/bigphysarea.c</td>
<td>Functions for boot-time reservation and allocation/deallocation</td>
</tr>
<tr>
<td>mm/Makefile</td>
<td>Modified to compile bigphysarea.c into the kernel</td>
</tr>
</tbody>
</table>

It should be apparent, from the i386-specific changes in this list, that other kernel modifications were necessary to make this patch work under an ARM architecture. One change I made was to arch/arm/config.in, to provide a configuration option for enabling the patch when compiling for an ARM environment.

Another change I made was to the kernel initialization code. While experimenting with the "mem" parameter on the ARM kernel, our group discovered that kernel command line parameters were becoming corrupted. These parameters are normally passed to the kernel by the BIOS, which writes them to an address specified in the kernel file's header. Somewhere, either in the BIOS or the kernel, this string was being corrupted.

This is significant because the Big Physical Area patch needs the size of the reserved memory pool, in pages, to be passed to the kernel at boot time. After a few failed
attempts at debugging the problem, I devised a workaround for the sake of expediency. This involved modifying init/main.c to call bigphysarea_setup() at the beginning of the kernel initialization process. This function is normally called automatically when the "bigphysarea=<p>" parameter is passed to the kernel, where <p> is the number of pages to reserve. It is passed to the function as a string, which consists of the text representation of <p>. My modification to init/main.c defines this string as follows:

```c
#define BIGPHYS_PAGES_STR "128" /* 512 KB */
```

This reserves 128 pages, which are 4K each, at boot time. This gives us a guaranteed shared memory window size of 256K, since up to 256K of the reserved pages must be wasted in order to achieve the proper alignment. However, this is still far better than the 768K that would have been wasted using the "mem=1M" parameter.

One important note about this patch is a legal one rather than a technical one. Because this patch, and the kernel it modifies, is licensed under the GNU General Public License, any compiled kernels that are distributed with my customized patch must come with the patch itself\(^1\). The open nature of the Linux kernel is what allowed our project to succeed in the first place, so this license should be respected.

4. The Shared Memory Device Drivers

---

\(^1\)See the file "COPYING " in the kernel source code, available in reference [4]
4.1. Purpose

The shared memory device drivers are intended to provide a communications channel between the host and the EBSA. Initially, they were planned to provide user-space access to the shared memory on both the host and the EBSA. The intent was to support the polling state-machine network simulation written by Bo Wu.

The Bo Wu simulation was originally written as a proof-of-concept for network communication via a shared memory channel. Since the shared memory device drivers did not exist when it was written, it simulated the shared memory on a single system using the standard SysV UNIX inter-process shared memory subsystem. This creates a shared memory channel between the host and EBSA simulation processes, rather than between the host and EBSA themselves [11].

In order for the Bo Wu simulation to support the shared memory device drivers, Rob McCready modified it to use the actual shared memory. This involved replacing the SysV shared memory setup code with the the `mmap()` system call, which provides shared memory between a user-space process and the PCI shared memory device drivers in the kernel.

4.2. Requirements
4.2.1. Host Shared Memory Device Driver (*hostpoll*)

Program the appropriate 21554 BAR with an arbitrary address in the 4 GB PCI address space.

The address must not be already used by another device

Linux handles this via its PCI function calls

Provide user-space access to the memory:

Creation of a device node (*/dev* file)

open() system call

close() system call

mmap() system call

4.2.2. EBSA Shared Memory Device Driver (*ebsapoll*)

Allocate the shared memory

Program the 21285 with the SDRAM address to be shared

Program the 21554 with the PCI address of the shared memory

Provide user-space access to the memory (same as in host driver)
5. First Implementation

Initially, only the driver on the EBSA side (ebsapoll) was written, since the feasibility of the entire CiNIC project, as planned, hinged on whether the 21285 and 21554 could be programmed to do what we need and whether the shared memory could be successfully allocated. The host driver (hostpoll) is comparatively much simpler and involves less risk, since it does the same things as any other driver in the Linux kernel for a PCI device. Thus, this chapter does not describe hostpoll.

Both the hostpoll and ebsapoll drivers are Linux kernel modules. This means that they are compiled into object files called hostpoll.o and ebsapoll.o, but not linked into executables. Linux kernel drivers such as these are not "run" like a normal Linux application or daemon, but instead are dynamically linked into the currently running kernel with the insmod command. They can also be safely unloaded from a running kernel with the rmmod command [9]. Additionally, it should be possible to statically link them into the kernel during the kernel build process. I have not tried to do so because that would lengthen the driver development cycle significantly, since the only way to load a new kernel is to reboot the system.
5.1. Memory Allocation

As mentioned previously, memory allocation is achieved via the Big Physical Area patch. This patch exports the following two functions to kernel modules:

```c
void bigphysarea_alloc_pages(int count, int align, int priority)
void bigphysarea_free_pages(caddr_t base)
```

`bigphysarea_alloc_pages()` takes as its parameters "count", which is the number of pages to allocate from the pool; "align" which is an alignment factor that describes the what multiple of pages the memory must start on; and "priority", which tells the allocation algorithm how hard it should try to free pages in order to meet the request. It returns the kernel virtual address of the beginning of the allocated region if it is successful, or NULL if it is unsuccessful.

I wrapped this function with a function in `ebsapoll`, called `alloc_mem()`. First, this function calculates the number of pages needed for the requested number of bytes. Next, it calls `bigphysarea_alloc_pages()` with "count" and "align" both equal to this number of pages, since the memory must be aligned on a multiple of its size. "priority" is set to GFP_KERNEL, which is the default for most kernel memory allocations. If the allocation was successful, the address is stored in a global
variable called "shm_addr". Finally, the memory is cleared with the `memset()` kernel function to prevent users from looking at the former contents of the memory, which could include passwords or the private data of other users.

`bigphysarea_free_pages()` takes as its single parameter the base kernel virtual address that was returned by `bigphysareaAlloc_pages()` at the time of allocation. The deallocation is always successful, and therefore it does not return anything. Like `bigphysareaAlloc_pages()`, I wrapped this function with another `ebsapoll` function, `free_mem()`, which simply checks that the global `shm_addr` variable has been set before trying to free it.

### 5.2. SDRAM Window Setup Code

While studying the EBSA BIOS code, Jim Fischer and I noticed that the 21285's SDRAM window was being set up near the beginning of the BIOS's hardware initialization. The initial CSR values were being set such that only the minimum possible amount of memory, 256 KB, was being mapped. We presumed that this was being done to minimize the chance of other code inadvertently corrupting memory, since if the SDRAM window were large, device probing routines would have a greater chance of accidentally writing to a PCI bus address that is mapped onto SDRAM.
Because the SDRAM window would already be set up by the time the Linux kernel boots, any kernel code that would reconfigure it would have to do so in a safe manner. The procedure for doing this is as follows:

1. **Make certain that PCI host controller is available:**
   
   Linux provides a function, `pcibios_present()`, which takes no arguments and returns TRUE if a PCI host controller is present [9]. Since the driver could be compiled and loaded under an x86 architecture with no PCI bus, this prevents the user from crashing the kernel with the PCI functions used in later steps.

2. **Find the 21285:**

   To further ensure that it is being loaded under the correct platform, the driver checks whether the 21285 is present. This is achieved with the following function call:
   ```c
   rc = pcibios_find_device(vid, did, index, &bus, &func)
   ``
   "vid" and "did" are the PCI vendor and device identifiers that are hard-coded into the 21285's configuration space. "index" is set to zero in this case, which indicates that the first matching device is to be returned. This is acceptable since only one 21285 will ever exist on the secondary bus. If the device is found, its PCI bus and function numbers are returned in the variables
pointed to by the final two parameters, and the function returns
PCIBIOS_SUCCESSFUL [9].

3. Clear the Initialize Complete bit:

This is bit 0 of the SA-110 Control Register (CSR offset 13Ch). Clearing it causes the 21285 to respond to PCI configuration cycles with a retry response [6]. This prevents the SDRAM window settings or other related settings from being modified while they are being set up, which could cause the 21285 to behave erratically. Additionally, the SDRAM Base Address Mask and SDRAM Base Address Offset registers are only writable via PCI when this bit is 1. Changing these registers through the SA-110 system bus (at memory offset 42000000h) is still allowed even when Initialize Complete is 0, and this is the method used to access these, and the rest of the CSRs, during SDRAM window setup.

To preserve the state of the other bits in the SA-110 Control Register, the readl() kernel function (read long; 32 bits) is used to store its contents in a local variable before it is modified. Bit 0 is cleared within this variable using standard C-language bit-operations, and the new value is written back to the register with the writel() kernel function.

4. Get the physical location of our SDRAM window

As mentioned previously, the Big Physical Area memory allocation function
returns a kernel virtual address. While this type of address makes it easy for
the kernel to access the memory, virtual addresses have no meaning in the
context of the 21285, the SDRAM, or the SA-110 bus. Therefore, the actual
physical memory address corresponding to the virtual address must be
determined so that it can be programmed into the 21285.

Linux makes this task easy with the virt_to_phys() kernel function [9].
It takes a void pointer as its argument, which represents the kernel virtual
address to convert, and returns the corresponding physical address as an
unsigned long. As one would guess, the actual conversion process varies with
the processor architecture and specific hardware platform. With the EBSA-
285 architecture, this process is as simple as subtracting PAGE_OFFSET from
the virtual address. PAGE_OFFSET defines the starting address where kernel
virtual addresses are mapped into both the kernel and user address spaces, and
is equal to C0000000h on most 32-bit platforms1.

5. Set the SDRAM Base Address Offset Register

This is a simple matter of writing the physical address obtained in the
previous step to the 21285 SDRAM Base Address Offset Register (CSR offset
104h.) The write1() kernel function is used again to write a 32-bit value,
although only bits 18 through 27 are significant. The other bits are read-only

1See the kernel file include/asm-arm/arch-ebsa285/memory.h in [4]
6. Set the Memory Space Enable, Memory Write and Invalidate Enable, and Fast Back-to-Back bits

These are bits 1, 4, and 9 of the 21285 PCI Command Register (CSR offset 04h), respectively [6]. They are set using the same `readl()`-modify-`writel()` procedure used for the Initialize Complete bit.

The values of the bits in this register have the same meaning as they do with any PCI device. The Memory Space Enable bit, when set to 1, allows the 21285 to respond to PCI memory transactions from other devices on the bus, including reads or writes to the mapped SDRAM. Setting the Memory Write and Invalidate Enable bit and the Fast Back-to-Back bit optimizes PCI efficiency.¹

7. Set the SDRAM Base Address Mask Register

This register determines the size of the SDRAM window, and is programmed into the 21554 SDRAM Base Address Offset Register (CSR offset 104h.) with the `writel()` function. Bits 18 through 27 constitute a size bit mask, and bit 31 disables the window altogether if it is set to 1. The rest of the bits are read-only and will always be zero [6].

¹The reasons for setting these bits are beyond the scope of this document. Please refer to [10] for more
The valid sizes that can be set and their corresponding masks are listed on page 7-28 of the 21285 Core Logic for SA-110 Microprocessor datasheet [6]. Initially, the driver will be hard-coded to a window size of 256 KB, so the corresponding mask value that will be programmed into this register is 00000000h.

8. Set the Initialize Complete bit

Once configuration is complete, this bit can be safely set back to 1. After this is done, the SDRAM-to-PCI translator will begin to respond to any valid PCI transaction, within the space defined by the SDRAM Base Address Mask and Offset registers.

No attempt was made to set up the 21554 in this stage of the driver's development, since this would require the host driver to exist in order to test it.

5.3. Additional Driver Functions

The EBSA shared memory driver implements the following functions in order to interface to the rest of the system:

5.3.1. ebsapoll_mmap()

This implements the mmap() system call, which allows a user-space process information.
to map its virtual address space onto kernel-space memory pages [9]. In this case, it allows the polling protocol software in user-space to transparently access the shared memory. The function first checks that the requested size is valid, and then uses the `remap_page_range()` kernel function to set up the page mapping.

5.3.2. **ebsapoll_open()**

The `open()` system call is used by user-space programs to create a new file descriptor, which can be used to access other driver functions such as `mmap()` [9]. This is handled automatically by the kernel, although the driver may explicitly override the implementation to do additional access control and initialization. In `ebsapoll`, this function increments a semaphore (open_lock) so that only one application may open it at a time, and invokes the `MOD_INC_USE_COUNT` macro to prevent the `ebsapoll` module from being unloaded while it is open.

5.3.3. **ebsapoll_close()**

The `close()` system call frees the file descriptor that was acquired with `open()` [9]. In `ebsapoll`, this function decrements the open_lock semaphore, and invokes the `MOD_DEC_USE_COUNT` macro so that the driver can be unloaded again.

5.3.4. **__init ebsapoll_probe()**

Functions with the `__init` qualifier are called automatically when the
module is loaded [2] (or during the boot process if the driver is compiled into
the kernel.) In ebsapoll, this function calls alloc_mem() to allocate the
shared memory, calls ebsapoll_pci_init() to program the 21554
registers, and calls the register_chrdev() kernel function to create a
device node with major number 253.

5.3.5. __exit ebsapoll_remove()

Since this function has the __exit qualifier, it is called automatically when
the module is unloaded [2]. It unregisters the device node, and calls
free_mem() to free the shared memory. It does not attempt to deprogram
the 21285’s memory window. Doing so is unnecessary as long as hostpoll is
unloaded before ebsapoll, to ensure that the shared memory is not accessed
across PCI again.

5.4. Test Results

To test the EBSA shared memory driver, I wrote a small program called "testapp".
This program opens the device node, gets a mapping of the shared memory with the
mmap() system call, performs an operation on the memory, closes it and unmaps it,
and then repeats the process a second time. The first time the mapping is performed,
the memory operation is a write of two arbitrarily chosen values to the first two bytes
of shared memory. During the second mapping, these values are read back from the
shared memory and printed out. The reason for having these two separate mappings is to ensure that the memory contents are being accessed via `mmap()` at a specific, fixed, location, and not at a different arbitrary address each time due to a faulty driver `mmap()` implementation.

Preliminary tests of the driver, with the SDRAM window code disabled, showed that the shared memory was most likely being accessed properly via the `mmap()` mechanism. The two bytes printed out from the mapped memory were correct, although there was no absolute guarantee that they were actually being written to the correct location.

To ensure this, I printed the same bytes directly from the shared memory at the beginning of the driver's `mmap()` function. At the first `mmap()` call, the values printed by the driver were both zero, as expected. At the second `mmap()` call, the values printed by the driver were the ones I wrote from user-space, during the first mapping. This proved that the `mmap()` implementation was correct and that the shared memory could be accessed successfully.

Results of the SDRAM window setup test were not so successful, however. The SDRAM window setup procedure appeared to work correctly since the values of the relevant CSRs, when read back and printed out, were correct. However, the system
would become unstable within a few seconds after the Initialize Complete was set.

The following two messages were printed to the system console repeatedly:

```
<3>eth0: Host error, FIFO diagnostic register 0000.
<6>eth0: using NWAY autonegotiation
```

These messages are from the ethernet card driver. Besides the 21285 and 21554, the ethernet card was the only other PCI device in the system at the time. Using the HP 16700A Logic Analyzer on the 21554's secondary PCI bus, I determined that a PCI device was trying to access a number of different addresses and failing. Checking these addresses against the values in the Base Address Registers of the ethernet card ,using the "lspci -vv" Linux command, I was able to confirm that these addresses were reserved for this card.

It seemed that reconfiguring the SDRAM window was somehow interfering with the normal operation of the PCI bus. However, some I/O transactions between the ethernet card and the 21285 still appeared to be working. Only transactions with addresses between E0000000h and EFFFFFFFh were failing.
6. Second Implementation

With this implementation, my main goals were to fix the problems with the SDRAM window and to set up the 21554 PCI bridge so that the host would be able to access the shared memory. The host driver was also written at this point, to allow this functionality to be tested.

The driver files, previously called *ebsapoll* and *hostdrv*, were renamed at this time to more clearly reflect what they specifically do. Since I decided not to integrate the Bo Wu polling state machine into these drivers as I had originally planned, I renamed them to *ebsamem* and *hostmem*.
6.1. Solutions to Previous Problems

From the previous results of my test of the SDRAM window setup code, it appeared that the ethernet card needs access to some memory resource that depends on the SDRAM window itself. My suspicion was that the card was trying to access the SDRAM through the 21285's window. Changing the window to point to a different location in memory would obviously cause the card to stop functioning properly if this were true.
This would make sense because bus-mastering Direct Memory Access (DMA) is a feature of many PCI network cards. For it to work, the card must be able to initiate PCI memory transactions that access the system's primary memory with no help from the CPU. The 21285 SDRAM window could be used for this, as long as it is large enough to provide access to every memory location that the card might want to read or write. If this was indeed the case, the SDRAM window would have to be initialized by the operating system sometime during the boot process.

To find out whether this was the case, I studied the Linux kernel source code. Specifically, I looked at the 21285 initialization code in `arch/arm/kernel/dec21285.c` [4] and found the following code in the `dc21285_init()` function:

```c
mem_size = (unsigned int)high_memory - PAGE_OFFSET;
for (mem_mask = 0x00100000; mem_mask < 0x10000000; mem_mask <<= 1)
  if (mem_mask >= mem_size)
    break;
*CSR_SDRAMBASEMASK = (mem_mask - 1) & 0x0ffc0000;
*CSR_SDRAMBASEOFFSET = 0;
```

In this code, the SDRAM mask and offset registers are being accessed directly, using pointers. The offset is being set to zero, which means that the window begins at the very start of SDRAM. It is much less obvious what the mask is being set to. `mem_size` is the amount of address space reserved for SDRAM, which is calculated...
by subtracting the beginning of kernel virtual address space (PAGE_OFFSET) from
the kernel virtual address where SDRAM ends (high_memory). The for loop then
determines whether mem_size is less than or equal to either 1 MB, 16 MB, or 256
MB. This determines the value of mem_mask, which is adjusted to meet the
requirements of the 21285 SDRAM Base Address Mask, and written to the
corresponding register.

Later on, this same function programs the SA-110 Command and PCI Control
registers, similarly to how I did it in ebsapoll's 21285 setup function. Once this
function completes, the hard work of setting up the SDRAM window has already
been done for us, and there is little reason to change it later.
6.2. Memory Map

Figure 5 shows the four different instances of addresses for the shared memory in the co-host. These are:

A kernel virtual address, which can be used for memory access from kernel code
A user virtual address which is mapped to physical memory via the mmap() call
A physical address in SDRAM
A PCI address, that is mapped onto the SDRAM via the 21285 SDRAM window
6.3. 21554 Setup

Although the shared memory is mapped onto the secondary PCI bus at this point, the host still cannot access it because it can only see devices on the primary PCI bus. The 21554 must be set up to pass memory transactions between the two busses. This setup procedure involves the following steps:

1. Set the size and properties of the SDRAM window

   This is done by writing the Downstream Memory 1 Setup Register (CSR offset B0h.) This can be performed by `ebsamem` with a PCI configuration transaction, but I have opted to set it up in the 21554’s serial ROM. Doing this ensures that the 21554 is always set up properly, even if the EBSA driver has not configured it yet. This is necessary since `hostmem` needs to know the size of the SDRAM window, regardless of whether the EBSA has finished booting or whether `ebsamem` has been loaded.

   The following table describes the contents of this register, as set by the serial ROM¹.

   \[
   \begin{array}{|c|c|c|}
   \hline
   \text{Bit} & \text{Name} & \text{Value} & \text{Notes} \\
   \hline
   0 & \text{Type Selector} & 0 & \text{Sets the region as memory space, instead} \\
   \hline
   \end{array}
   \]

   ¹See table 7-35 in [7] for additional documentation on this register
<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:1</td>
<td>Type</td>
<td>00</td>
<td>Indicates that the region may be located anywhere in PCI address space</td>
</tr>
<tr>
<td>3</td>
<td>Prefetchable</td>
<td>1</td>
<td>Memory may be cached within the 21554, and in intermediate PCI bridges</td>
</tr>
<tr>
<td>5:4</td>
<td>Reserved</td>
<td>00</td>
<td>Read-only; always zero</td>
</tr>
<tr>
<td>30:6</td>
<td>Size</td>
<td>11111111111111000000000000000</td>
<td>If a bit is 1, the corresponding bit in the Downstream Memory 1 BAR may be set. The size, in bytes, is equal to ((~Size + 1) * 2^6) The size in this example is 256K.</td>
</tr>
<tr>
<td>31</td>
<td>BAR_Enable</td>
<td>1</td>
<td>Enables the Downstream Memory 1 BAR</td>
</tr>
</tbody>
</table>

2. Determine the PCI address of the shared memory on the secondary bus

This is one of the most confusing aspects of the ebsamem driver. The goal is not only to find the PCI address where the SDRAM window begins, but also to find the PCI address that corresponds to where the driver's memory was allocated.

The PCI address where the SDRAM window begins is stored in the 21285 PCI SDRAM Base Address Register (CSR offset 18h.) Bits 0 through 3 are the standard PCI Type Selector, Type, and Prefetchable flags, as listed in Table V, so the actual PCI address is:

\[ \text{CSR\_PCISDRAMBASE \& 0xffffffff0} \]

The offset of the shared memory from the beginning of SDRAM is simply the
physical address of the shared memory. As mentioned previously, this is obtained with the \texttt{phys\_to\_virt()} function, which does the same thing as subtracting \texttt{PAGE\_OFFSET} from the virtual address. To obtain the PCI address of the shared memory, this offset is added to the PCI address of the beginning of the SDRAM window. Surprisingly, the ARM Linux kernel already provides a function called \texttt{__virt\_to\_bus()} that does all of this. The source code [4] for it contains the following single statement, as expected:

\begin{verbatim}
return ((res - PAGE_OFFSET) 
    + (*CSR_PCISDRAMBASE & 0xffffffff0));
\end{verbatim}

3. Program this address into the 21554

In order for the 21554 to know what address on the secondary bus it must use to access the shared memory, the PCI address of the shared memory found in the previous step is programmed into the "Downstream I/O or Memory 1 Translated Base Register" (CSR offset 98h) [7]. This only requires a single PCI configuration transaction. A function that does this, \texttt{i21554\_cohost\_config()}, was provided by Jason Hatashita in his \texttt{21554.c} file.\footnote{21554.c is compiled to 21554.o, which is linked with ebsamem.o and hostmem.o to form ebsamem_p.o and hostmem_p.o} This file contains many other 21554 helper functions, which
will be mentioned later in this document.

4. Assign a PCI address to the window on the primary bus

In order for the host to access the shared memory, the 21554 must be assigned a PCI address on the primary bus. PCI devices are automatically scanned for BARs at boot time, and if the BARs are enabled, unique PCI addresses are allocated and written into them. Fortunately, the 21554 Downstream Memory 1 BAR (CSR offset 18h) was enabled by the serial ROM, as explained in step one of this procedure, so the BAR will already be configured by the time hostmem is loaded.

6.4. Host Driver

The host driver (hostmem) is almost identical to ebsamem, except that it does not need to allocate any shared memory or set up the 21554. It provides the same open(), close(), and mmap() system call functionality that ebsamem does. The most significant difference is how it gets the address of the shared memory.

Getting the address of the shared memory is relatively easy. 21554.c contains a helper function, ui21554_host_config(), that returns the PCI address of the shared memory on the primary bus. This is done by reading the address from the 21554 Downstream Memory 1 BAR, and masking off the flags that reside in the
lowest four bits.

Since kernel code cannot access the shared memory by its PCI address, a kernel virtual mapping of the shared memory must be created. This is done with the \texttt{ioremap()} kernel function. This function takes a physical address and a size argument, creates the page tables for it, and returns the corresponding kernel virtual address [9]. This address is stored in the \texttt{shm_addr} global variable in \texttt{hostmem.c}.

6.5. Additional Changes in the Second Version

A number of small improvements were made to the driver since the first version. The following subsections describe these improvements.

6.5.1. Preliminary Interrupt Code

In preparation for the next generation of the driver, I have added some basic interrupt code to both \texttt{ebsamem.c} and \texttt{hostmem.c}. This consists of interrupt registration code, an interrupt handler, and an interrupt trigger ioctl.

The interrupt registration code uses the \texttt{iGet_irq_info()} function from \texttt{21554.c} to get the 21554's IRQ number. This is done by reading the "Interrupt Line" byte in the 21554's CSR space, at offset 3Ch. This function also obtains a unique 32-bit device identifier number that is used to identify this particular 21554, in
case more than one is using the same IRQ. The IRQ number and unique identifier are passed to the `request_irq()` kernel function, along with a pointer to the interrupt handler function.

After being registered with the `request_irq()` function, the interrupt handler is called automatically whenever an interrupt arrives. Since interrupts are not yet used for anything, the handler simply prints out a message and clears the interrupt condition with the `iClearHostINT()` function (in the case of `hostmem`), or the `iClearCoHostINT()` function (in the case of `ebsamem`). These functions are also provided by `21554.c`.

In order to test whether the interrupt code works, I decided to implement the `ioctl()` system call in `hostmem` and `ebsamem`. This allows a test application, called `int_test.c`, to notify the driver to send an interrupt across the 21554 by issuing the `CINIC_IOC_INTERRUPT` ioctl. The ioctl takes a single 16-bit unsigned argument. The drivers' ioctl handlers pass this argument to the `iInterrupt_cohost()` function (in the case of `hostmem`), or the `iInterrupt_host()` function (in the case of `ebsamem`), which trigger the interrupt and store the argument in the 21554's doorbell register. The system that receives the interrupt can query the doorbell register, and use its value to decide what action to take.
6.5.2. Kernel Access to Shared Memory

The Bo Wu user-space simulation was only designed for testing purposes. Networking calls on the host must be intercepted within the kernel, which means that the host's shared memory communication protocol code must also be in the kernel. Additionally, it is desirable to put the EBSA shared memory communication protocol code into the kernel in order to eliminate the overhead of making networking calls from user-space. Therefore, I had to implement a clean way for kernel code to access the shared memory.

Kernel shared memory access is provided in the ebsamem driver by the get_ebsa_shrmem() function. It passes by reference the size and kernel virtual address of the shared memory region to the calling function. It also invokes MOD_INC_USE_COUNT to prevent the module from being unloaded while the memory is in use. Once the memory is not needed any more, the release_ebsa_shrmem() function is called, which simply invokes MOD_DEC_USE_COUNT. The host driver provides two similar functions for the same purpose.

6.5.3. PCI Hotplug

The 2.4 version of the Linux kernel provides an optional new programming interface
for PCI devices. Previously, it was the driver's job to probe for devices upon initialization. However, some PCI host controllers allow devices to be attached while the system is already running, a feature called "PCI hotplug." This means that PCI drivers now need to tell the OS what devices they can handle, and provide a callback to notify the driver when a matching device is found.

The hostmem and ebsamem drivers both implement this new interface. During their initialization, they call the pci_module_init() function, which is passed a pci_driver structure. This structure contains the name of the driver, a pointer to a table of PCI identifiers to match, and pointers to "probe" and "remove" functions, which are called when the device is attached or removed. The probe function is passed a pointer to a pci_dev structure, which is used to identify the device when calling other PCI functions. This eliminates the need to refer to devices by their bus and function numbers.

6.5.4. Devfs Support

Another new feature of the 2.4 kernel is the Device Filesystem, which is known as devfs. This is a "virtual filesystem" that contains dynamically generated device nodes. Normally, device nodes are manually created by the user in the /dev directory, with the mknod command. This is inconvenient with dynamically attachable devices such as USB and PCI devices, and it requires that drivers reference the device nodes with
major and minor numbers. Since devfs creates the devices on-the-fly and references them by name, these problems are averted.

The `ebsamem` and `hostmem` drivers implement this feature by using the `devfs_register_chrdev()` and `devfs_unregister_chrdev()` functions instead of the `register_chrdev()` and `unregister_chrdev()` functions. These new functions take the same arguments as the old ones, and if devfs is not enabled in the kernel, they behave just like the old functions.

### 6.6. Test Results

#### 6.6.1. Basic Shared Memory Test

To test whether communication through the 21554 was working, I created a modified version of `testapp.c` for the host, called `testapp_host.c`. It only performs one mapping, during which it writes data to bytes 2 and 3 of the shared memory, and prints out bytes 0 and 1. I then "inverted" this process for the EBSA test app, which I called `testapp_ebsa.c`. This program writes data to bytes 0 and 1 of the shared memory, and prints out bytes 2 and 3.

When I ran `testapp_host` for the first time, after loading the shared memory device drivers, it displayed:
mem[0] = 0, mem[1] = 0
Should be 95 and 46

Since testapp_ebsa had not been run yet, bytes 0 and 1 had not been modified yet. This is the expected result, and is a good test of whether the memory had been properly initialized to zero.

Next, I ran testapp_ebsa, which displayed:

mem[0] = 13, mem[1] = 37
Should be 13 and 37

This proved that writes to shared memory addresses by the host were actually being stored in the shared memory on the EBSA. Next, I ran testapp_host again to see if the bytes that were modified by testapp_ebsa could be read by the host. The results it displayed proved that this was the case:

mem[0] = 95, mem[1] = 46
Should be 95 and 46

6.6.2. Bo Wu Simulation Test

In order to test the drivers more thoroughly, I ran the Bo Wu simulation that had been
modified to use the `mmap()` system call.

First, the shared memory device drivers were loaded:

- On the EBSA, I executed "`insmod ebsamem_p.o`".
- On the host, I executed "`insmod hostmem_p.o`".

Next, the low-level communication layer was started:

- On the EBSA, I executed "`ebsacom`".
- On the host, I executed "`hostcom`".

Next, the network layer must was started:

- On the EBSA, I executed "`ebsaapp`".

On a third system on the local network, which I will designate the "receiver", I executed "`udp_recv`". This displays the port number that the receiver is listening on, and waits for UDP packets to arrive.

On the host, I executed "`hostapp <hostname> <port>`" using the hostname of the receiver and the port number that was displayed when `udp_recv` was started.

When I typed a line of text into `hostapp` and pressed the <Enter> key, the message was displayed on the receiver. I did this multiple times, and it usually worked.
However, in some cases the communication layer would stop responding. Rob McCready determined that there were race conditions in the communication layer's state machine, that were causing the state machine on the host to become out-of-sync with the EBSA.

After fixing these bugs the simulation worked well, although it usually took at least one second to transfer only a few bytes of network payload. This is not surprising, considering the inherent inefficiencies of polling state variables repeatedly. The data has to travel across two PCI busses and through the 21554 to get to the host, so any inefficiencies are multiplied by this added latency as well.

6.6.3. FTP Test

To improve network performance and functionality above what Bo Wu's code was providing, Rob McCready wrote a kernel-mode communication layer that intercepts standard socket calls on the host. The calls are encoded and sent to the EBSA via the shared memory device drivers. The EBSA then decodes them and sends them to the remote host. Finally, the EBSA re-encodes any replies that it receives and sends them back to the host.

As a proof-of-concept, the CiNIC group decided to implement support for the FTP protocol and test it. As our test platform, we used the Debian FTP client running on
the host and the WU-FTPD server running on another Linux system.

The test procedure was as follows:

EBSA: "insmod ebsamem_p.o"
Host: "insmod hostmem_p.o"
EBSA: "insmod ebsa_p.o"
Host: "insmod host_p.o"
Host: Run deb-ftp with the IP address of the FTP server, and send a file to the server.

The file was successfully transferred at speeds varying from 200KB to 800 KB per second. This is many orders of magnitude faster than the data transfers were with the Bo Wu simulation.

6.6.4. Interrupt Test

To test whether interrupts could be sent between the host and the EBBA, I wrote a small test application called "int_test". This application takes as its first argument the name of the shared memory driver's device node, and as its second argument an unsigned 16-bit integer to be stored in the 21554's doorbell register. The test application functions exactly the same on the host and on the EBBA, by issuing the CINIC_IOC_INTERRUPT ioctl on the specified device node. This causes the driver
to send an interrupt across the 21554 to the other system.

I began the test by running `int_test` on the host:

```
# int_test /dev/hostmem 31337
```

In the system log (`/var/log/messages`) on the EBSA, the following message appeared:

```
ebsa285: ebsamem: Got interrupt
```

This is the message that `ebsamem` prints when its interrupt handler is called. This proves that the interrupt successfully made it across both PCI busses and through the 21554. I also tested sending an interrupt from the EBSA to the host. This was successful as well.

One caveat, however, is that neither driver is able to read the contents of the doorbell register yet. This should be as simple as adding a single Type 0 configuration read function call to the interrupt handler, though.
7. Conclusion

Both the host and EBSA shared memory drivers are completely implemented and functional, for the purposes of supporting a polling data transfer protocol. A minimum of interrupt functionality has been implemented and successfully tested as well. However, more work will have to be done to integrate the triggering and handling of interrupts with kernel code outside of my drivers. This will involve writing an interrupt handler registration layer that other kernel modules can use to hook into my drivers’ interrupt handlers. Additionally, functions will need to be exported from my drivers to allow other drivers to send interrupts.

While interrupt support in the communication protocol will reduce unnecessary PCI bus utilization, support for bus-mastering direct memory access (DMA) will be necessary to reduce processor utilization to a minimum. This will require further research into the capabilities and limitations of the 21285 chipset and the 21554 bridge.

Eventually, the drivers will need to be modified or rewritten to work under a newer hardware platform that is more powerful, and has a lower per-unit cost, than our current one. Some prospective platforms are the StrongARM SA-1100, or the Altera Excalibur. Hopefully, this paper has provided enough insight into the process of supporting a shared memory architecture to make such a transition easier.
Works Cited


Appendix A - Source Code of First Implementation

common.h

******************************************************************************
* Header file for EBSA communications driver *
* Author: Mark McClelland (3Com/Cal Poly Project) *
* Date: 10/3/2000
******************************************************************************

#ifndef __LINUX_EBSA_COMMON_H
#define __LINUX_EBSA_COMMON_H

/* Enable debugging code */
#define DEBUG 1

/* Device node major and minor numbers */
#define EBSA_MAJOR 253
#define EBSA_MINOR 0

/* Size of shared memory segment in bytes. Must be multiple of PAGE_SIZE */
#define SHM_SIZE 262144

/* 21285 PCI Vendor and Product IDs */
#define EBSA285_VID 0x1011
#define EBSA285_PID 0x1065

/* Memory offset of SA-110 CSR space */
#define SA110_CSR 0x42000000

/* SDRAM Base Address Mask. (Set to 256KB) */
#define SA110_SDRAM_BA_MASK 0x00000000
#define SA110_SDRAM_BA_OFFSET 0x100

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

#endif
ebsapoll.c

/**************************************************************************
* EBBA communications driver (polling version)
* 
* Author: Mark McClelland (3Com/Cal Poly Project)
* Date: 10/3/2000
**************************************************************************/

#include <linux/kernel.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/init.h>
#include <linux/ctype.h>
#include <linux/smp_lock.h>
#include <linux/fs.h>
#include <linux/vmalloc.h>
#include <linux/slab.h>
#include <linux/pagemap.h>
#include <linux/pci.h>
#include <asm/semaphore.h>
#include <linux/wrapper.h>
#include <linux/bigphysarea.h>
#include "common.h"

MODULE_AUTHOR("Mark McClelland (3Com/Cal Poly Project)"");
MODULE_DESCRIPTION("EBBA communications driver (polling version)"");

/**************************************************************************
* Global variables
**************************************************************************/

static const char version[] = "0.06";

/* Kernel Virtual Address of shared memory */
void * shm_addr;

/* Prevents multiple simultaneous open()'s */
struct semaphore open_lock;

/**************************************************************************
* Memory allocation functions
**************************************************************************/

/* Allocate shared memory. */
static int alloc_mem(void)
{
    /* Determine number of pages needed */
    int shm_pages = SHM_SIZE / PAGE_SIZE;

    /* Allocate a chunk of memory from the pool. Returns a Kernel
/* Virtual Address. */
shm_addr = (void *) bigphysarea_alloc_pages(shm_pages,
            shm_pages,
            GFP_KERNEL);

/* Was the allocation successful? */
if (shm_addr == NULL) {
    return -ENOMEM;
} else {
    printk("ebsapoll: Allocated %d pages at va=%p, pa=%p\n",
           shm_pages,
           shm_addr,
           (void *)virt_to_phys(shm_addr));
}
return 0;

/* Free shared memory. */
static void free_mem(void)
{
    if (!shm_addr)
        return;
    bigphysarea_free_pages(shm_addr);
}

 /******************************************************************
 * System calls
 *******************************************/
static int ebsa_open(struct inode *inode, struct file *file)
{
    /* Prevent multiple simultaneous opens */
    down(&open_lock);

    printk(KERN_INFO "ebsa: open\n");

    /* Prevent module from being unloaded while open */
    MOD_INC_USE_COUNT;

    return 0;
}

static int ebsa_close(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "ebsa: close\n");

    /* Allow module to be unloaded */
    MOD_DEC_USE_COUNT;

    /* Allow the next open() to succeed */
    up(&open_lock);

    return 0;
}
/ * Code partially taken from linux/drivers/usb/ov511.c by Mark McClelland and from "Linux Device Drivers" by Alessandro Rubini */
static int ebsa_mmap(struct file *file, struct vm_area_struct *vma)
{
    /* Starting physical address of SDRAM region to map */
    unsigned long phys_start;

    /* Requested size of mapping */
    unsigned long size = vma->vm_end - vma->vm_start;

    /* Requested offset of mapped area from beginning of shared * memory */
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

    if (shm_addr == NULL)
        return -ENOMEM;

    /* Did the user ask for too much? */
    if (size > SHM_SIZE)
        return -EINVAL;

    /* Give warning if too little was mapped */
    if (size != SHM_SIZE)
        printk(KERN_INFO "ebsapoll (mmap): warning: \n requested size does not match SHM_SIZE\n");

    /* Calculate the starting address, based on the offset */
    phys_start = (unsigned long) shm_addr + offset;

    printk("remap_page_range(0x%lx, 0x%lx, %ld, ...)
",
      vma->vm_start,
      virt_to_phys((void*)phys_start),
      size);

    /* Do the page mapping */
    if (remap_page_range(vma->vm_start,
                         virt_to_phys((void*)phys_start),
                         size,
                         vma->vm_page_prot)) {
        printk("remap_page_range() failed\n");
        return -EAGAIN;
    }

    return 0;
}

/* System call table */
struct file_operations ebsa_fops = {
    open:     ebsa_open,
    release:  ebsa_close,
    mmap:     ebsa_mmap,
};

/*********************************************************************************/
* I/O functions
*
*******************************************************************/

/* Sets bits in the specifies 21285 CSR.
* Parameters:
* offset - Byte offset of register from beginning of CSR space
* mask - If bits are set, set the corresponding CSR bits
*/
static void CSR_set_bits(unsigned long offset, unsigned long mask)
{
    unsigned long oldval, newval, ioaddr;
    /* Get address of register */
    ioaddr = SA110_CSR + offset;
    /* Save the current contents */
    oldval = readl(ioaddr);
    /* Compute the new value */
    newval = oldval | mask;
    /* Write new value to register */
    writel(newval, ioaddr);
    #ifdef DEBUG
    printk("addr 0x%lx: old=0x%lx target=0x%lx actual=0x%lx\n",
        ioaddr,
        oldval,
        newval,
        readl(ioaddr));
    #endif
}

/* Clears bits in the specifies 21285 CSR.
* Parameters:
* offset - Byte offset of register from beginning of CSR space
* mask - If bits are set, clear the corresponding CSR bits
*/
static void CSR_clear_bits(unsigned long offset, unsigned long mask)
{
    unsigned long oldval, newval, ioaddr;
    /* Get address of register */
    ioaddr = SA110_CSR + offset;
    /* Save the current contents */
    oldval = readl(ioaddr);
    /* Compute the new value */
    newval = oldval & ~mask;
    /* Write new value to register */
    writel(newval, ioaddr);
    #ifdef DEBUG
    printk("addr 0x%lx: old=0x%lx target=0x%lx actual=0x%lx\n",
        ioaddr,
oldval,
newval, 
readl(ioaddr));

#endif

/******************************************************************************
 * Init functions and module functions
 * ******************************************************************************/

/* Sets up the 21285 shared memory window */
static int ebsapoll_pci_init(void)
{
  unsigned long phys_addr;
  unsigned char ebsa285_bus, ebsa285_func;
  int result;

  /* Sanity check - make sure PCI host controller is available */
  if (!pcibios_present()) {
    printk(KERN_ERR "ebsapoll: PCI is not configured or not \n available\n");
    return -ENODEV;
  }

  /* Find the 21285's bus and function numbers */
  result = pcibios_find_device(EBSA285_VID, /* Vendor */
                              EBSA285_PID, /* Product */
                              0, /* Device index */
                              &ebsa285_bus, /* Bus */
                              &ebsa285_func); /* Function */

  if (result != PCIBIOS_SUCCESSFUL) {
    printk(KERN_ERR "ebsapoll: Could not find EBSA285\n");
    return -ENODEV;
  }

  /* Prevent rest of kernel from executing, for extra safety */
  lock_kernel();

  /* Clear the Initialize Complete bit */
  printk("Clearing the Initialize Complete bit\n");
  CSR_clear_bits(SA110_REG_CONTROL, 0x00000001);

  /* Get the physical location of our SDRAM window */
  phys_addr = virt_to_phys(shm_addr);

  /* Set the SDRAM base address offset. */
  printk("Setting SDRAM offset to %lx\n", phys_addr);
  writel(phys_addr, SA110_CSR + SA110_REG_SDRAM_BA_OFFSET);

  /* Set the Memory Space Enable, Memory Write and Invalidate, and
   * Fast Back-to-Back bits */
  printk("Setting command reg bits\n");
  CSR_set_bits(SA110_REG_COMMAND, 0x00000212);

  /* Set the SDRAM base address mask. */
printk("Setting SDRAM mask to 0x%lx\n", SA110_SDRAM_BA_MASK);
write1(SA110_SDRAM_BA_MASK, SA110_CSR + SA110_REG_SDRAM_BA_MASK);

/* Set the Initialize Complete bit */
printk("Setting the Initialize Complete bit\n");
CSR_set_bits(SA110_REG_CONTROL, 0x00000001);
unlock_kernel();
return 0;
}

/* Main entry point. Called when kernel boots (or if built as a
module, by init_module() upon module load.) */
static int __init ebsapoll_init(void)
{
    int rc;

    printk(KERN_INFO "EBSA polling communication driver (ebsapoll) \n
    version %s initializing\n", version);

    /* Initialize the open lock to an unlocked state */
    init_MUTEX(&open_lock);

    /* Allocate the shared memory */
    rc = alloc_mem();
    if (rc < 0) {
        printk(KERN_ERR "ebsa: Memory init failed\n");
goto out;
    }

    /* Set up the 21285 shared memory window */
    rc = ebsapoll_pci_init();
    if (rc < 0) {
        printk(KERN_ERR "ebsa: PCI init failed\n");
goto out_dealloc;
    }

    /* Create the device node */
    rc = register_chrdev(EBSA_MAJOR, "ebsa", &ebsa_fops);
    if (rc < 0) {
        printk(KERN_ERR "ebsa: can't get major %d\n", EBSA_MAJOR);
goto out_dealloc;
    }

    return 0;

out_dealloc:
    free_mem();
out:
    return rc;
}

/* Clean up, in preparation for module unload */
static void __exit ebsapoll_exit(void)
{
    printk(KERN_INFO "EBSA polling communication driver (ebsapoll) \n
    exiting\n");
if (unregister_chrdev(EBSA_MAJOR, "ebsa") < 0) {
    printk(KERN_ERR "ebsa: can't unregister dev on major %d",
            EBSA_MAJOR);
}

    free_mem();
}

/* Module entry/exit points */
#endif

#define MODULE

int init_module(void)
{
    return ebsapoll_init();
}

cleanup_module(void)
{
    ebsapoll_exit();
}
#endif
testapp.c

/******************************************************************************
 * Test application for ebsapoll driver
 *
 * Does the following:
 * 1. Creates a user-space mapping of the shared memory
 * 2. Modifies the memory
 * 3. Unmaps the memory
 * 4. Creates a second mapping of the shared memory
 * 5. Prints the previously modified bytes
 * 6. Unmaps the shared memory
 *
 * Author: Mark McClelland (3Com/Cal Poly Project)
 * Date: 10/3/2000
*******************************************************************************/

#include <stdio.h>
#include <stddef.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "common.h"

int main(void)
{
    int fd1, fd2, rc;
    unsigned char * mem1;
    unsigned char * mem2;
    unsigned long memsize = SHM_SIZE;

    /* Open the device node */
    fd1 = open("/dev/ebsapoll", O_RDWR);
    if (fd1 < 0) {
        perror("open");
        return 1;
    }

    /* Create the first mapping. It is created with the MAP_SHARED 
     * option so that the changes are permanent stored in the 
     * shared memory rather than in a local copy */
    mem1 = mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd1, 0);
    if (mem1 == NULL) {
        perror("mmap (1)"),
        exit(1);
    }

    /* Modify the memory */
    mem1[0] = 95;
mem1[1] = 46;
/* Close the device node */
close(fd1);

/* Destroy the shared memory mapping */
rc = munmap(mem1, SHM_SIZE);
if (rc) {
    perror("munmap");
    exit(1);
}

/* Reopen the device node */
fd2 = open("/dev/ebsapoll", O_RDWR);
if (fd2 < 0) {
    perror("open");
    return 1;
}

/* Create the second mapping */
mem2 = mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd2, 0);
if (mem2 == NULL) {
    perror("mmap (1)");
    exit(1);
}

/* Print out the bytes that we had previously changed */
printf("mem2[0] = %d, mem2[1] = %d\nShould be 95 and 46\n",
       mem2[0], mem2[1]);

/* Close the device node */
close(fd2);

/* Destroy the shared memory mapping */
rc = munmap(mem2, SHM_SIZE);
if (rc) {
    perror("munmap");
    exit(1);
}

return 0;
Appendix B - Source Code of Second Implementation

global.h

/*
global.h - Cal Poly 3Com CiNIC project

Global configuration variables and debug macros
*/
#ifdef _GLOBAL_H_
#define _GLOBAL_H_
#include <linux/kernel.h> /* KERN_DEBUG, KERN_INFO */

/*
Sets the size of the shared memory segment.
This must be a multiple of PAGE_SIZE, and must exactly match
the size set in the 21554 exactly (for proper alignment)
*/
#ifdef SIMULATION
#define SHRMEM_SIZE 65536 /* 64k */
#else
#define SHRMEM_SIZE 262144 /* 256k */
#endif

typedef struct {
    unsigned long shared_mem_addr;
    unsigned long shared_mem_size;
    unsigned int status;
} module_info_t;

/*
module_info_t status codes
0 - Running
1 - No more new sockets sent to ebsa
2 - system call layer unloaded
3 - Forced unload of syscall layer and MOD_USE_COUNT
*/

/*
DEBUG_LEVEL
0 - production
1 - major stuff
2 -
9 - massive proto loggin'
*/
#define DEBUG_LEVEL 8
#define PRINT_DEBUG(l, a...) \
    if (l <= DEBUG_LEVEL) { \ 
        printk(KERN_DEBUG a); \ 
    }
#define PRINT_ERROR(a...) printk(KERN_INFO a)
#endif
ebsamem.h

/******************************************************************************
* EBSA shared memory setup driver header file                                *
*                                                                          *
* Author: Mark McClelland (3Com/Cal Poly Project)                         *
* Initial Date: 10/3/2000                                                  *
*******************************************************************************/

#ifndef __LINUX_EBSAMEM_H
#define __LINUX_EBSAMEM_H

/* Debug level. 1 is least, 5 is greatest. Undefine to disable. */
#define EBSAMEM_DEBUG 1

/* Debug printk() macro */
#ifdef EBSAMEM_DEBUG
#define PDEBUG(level, fmt, args...) \
    if (EBSAMEM_DEBUG >= level) printk("ebsamem: " fmt "\n", ## args)
#else
#define PDEBUG(level, fmt, args...) do {} while(0)
#endif

/**************************
* Device node major and minor numbers                                     *
#define EBSA_MAJOR 253
#define EBSA_MINOR 0

/* Send an interrupt. This is the same on the host and the EBSA */
#define CINIC_IOC_INTERRUPT _IOW (0xAA, 128, u16)

#define EBSA285_VID 0x1011
#define EBSA285_PID 0x1065
#endif /* ifndef __LINUX_EBSAMEM_H */
ebsamem.c

/**
 * EBSA shared memory allocation and access
 * Requires bigphysarea kernel patch to successfully compile
 * 
 * Author: Mark McClelland (3Com/Cal Poly Project)
 * Initial Date: 10/3/2000
 */

#include <linux/kernel.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/init.h>
#include <linux/ctype.h>
#include <linux/smp_lock.h>
#include <linux/fs.h>
#include <linux/vmalloc.h>
#include <linux/slab.h>
#include <linux/pagemap.h>
#include <linux/pci.h>
#include <asm/semaphore.h>
#include <linux/wrapper.h>
#include <linux/bigphysarea.h>
#include <linux/interrupt.h>
#include <linux/devfs_fs_kernel.h>
#include <asm/uaccess.h>
#include "global.h"
#include "ebsamem.h"
#include "21554.h"

#define ENABLE_INTERRUPT_CODE

MODULE_AUTHOR("Mark McClelland (3Com/Cal Poly Project)");
MODULE_DESCRIPTION("EBSA shared memory driver");

/**
 * Global variables
 */

/* Driver version string */
static const char version[] = "0.14";

/* IRQ number of 21554 */
static unsigned int irq = -1;

/* Unique identifier for our particular 21554 */
static u32 dev_id;

/* Kernel Virtual Address of shared memory */
static void * shm_addr;

/* PCI address in EBSA-285's SDRAM BAR */
static unsigned long ebsa_pci_addr;

/* Prevents multiple simultaneous open()'s */
static struct semaphore open_lock;

/* Indicates that IRQ was successfully registered */
static int got_irq;

/* The 21554 PCI device struct */
static struct pci_dev *g_pcidev;

/****************************************************************************
*/
/* Memory allocation functions. These require the bigphysarea patch. Since
* the ARM kernel is not accepting kernel parameters properly, the number of
* bigphysmem pages *(2 *(SHRMEM_SIZE >> PAGE_SHIFT)) must be hard coded into the
* kernel init code.
*/
/****************************************************************************

/* Allocate shared memory. Memory has already been reserved at boot time; we
just need to reserve a piece of it. */
static int alloc_mem(void)
{
    /* Determine number of pages needed */
    int shm_pages = SHRMEM_SIZE / PAGE_SIZE;

    /* Allocate a chunk of memory from the pool. Returns a Kernel Virtual Address. */
    shm_addr = (void *) bigphysarea_alloc_pages(shm_pages, shm_pages, GFP_KERNEL);

    /* Was the allocation successful? */
    if (shm_addr == NULL)
        return -ENOMEM;
    else
        PDEBUG(1, "Allocated %d shared memory pages at va=%p, \ pa=%p\n", shm_pages, shm_addr, (void *)virt_to_phys(shm_addr));

    /* Clear it out - failure to do so may be a security risk */
    memset(shm_addr, 0, SHRMEM_SIZE);

    return 0;
}

/* Free shared memory. */
static void free_mem(void)
{
    if (!shm_addr)
        return;

    bigphysarea_free_pages(shm_addr);
}
/******************************************
* System calls - These provide user-space access to the shared memory when needed.
******************************************/

static int ebsamem_open(struct inode *inode, struct file *file)
{
    /* Only allow one process at a time */
    down(&open_lock);

    printk(KERN_INFO "ebsa: open\n");

    /* Prevent module from being unloaded while open */
    MOD_INC_USE_COUNT;

    return 0;
}

static int ebsamem_close(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "ebsa: close\n");

    /* Allow module to be unloaded */
    MOD_DEC_USE_COUNT;

    /* Allow the next open() to succeed */
    up(&open_lock);

    return 0;
}

/* This provides the user a window onto the shared memory buffer. */
/* Code partially taken from linux/drivers/usb/ov511.c by Mark McClelland and from "Linux Device Drivers" by Alessandro Rubini */

static int ebsamem_mmap(struct file *file, struct vm_area_struct *vma)
{
    /* Starting physical address of SDRAM region to map */
    unsigned long phys_start;

    /* Requested size of mapping */
    unsigned long size = vma->vm_end - vma->vm_start;

    /* Requested offset of mapped area from beginning of shared memory */
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

    if (shm_addr == NULL)
        return -ENOMEM;

    /* Did user ask for too much? (Too little is OK) */
    if (size > SHRMEM_SIZE)
        return -EINVAL;
/* Give them a warning if they asked for too little, though */
if (size != SHRMEM_SIZE)
PDEBUG(1, "warning: requested size does not match \nSHRMEM_SIZE\n");

/* Calculate the starting address, based on the offset */
phys_start = (unsigned long) shm_addr + offset;
PDEBUG(1, "remap_page_range(0x%lx, 0x%lx, %ld, ...)\n",
       vma->vm_start, virt_to_phys((void*)phys_start), size);

/* Do the page mapping */
if (remap_page_range(vma->vm_start,
                    virt_to_phys((void*)phys_start),
                    size,
                    vma->vm_page_prot))
{
    printk(KERN_ERR "remap_page_range() failed\n");
    return -EAGAIN;
}
return 0;

static int ebsamem_ioctl(struct inode *inode,
struct file *file,
unsigned int cmd,
unsigned long arg)
{
switch(cmd) {
    case CINIC_IOC_INTERRUPT:
    {
#ifdef ENABLE_INTERRUPT_CODE
    u16 doorbell_val;
    /* Copy the data from user-space */
    if (copy_from_user(&doorbell_val,
                      (void *)arg,
                      sizeof(doorbell_val)))
        return -EFAULT;
    PDEBUG(1, "Sending interrupt to host (doorbell=%d)",
           doorbell_val);
    /* Send the interrupt */
    if (iInterrupt_host(g_pcidev, doorbell_val))
        return -EIO;
#endif
    return 0;
    }
    default:
    {
        return -EINVAL;
    }
}
return 0;
/* System call table */
static struct file_operations ebsa_fops = {
  open: ebsamem_open,
  release: ebsamem_close,
  mmap: ebsamem_mmap,
  ioctl: ebsamem_ioctl,
};

/******************************************************************
* * Interrupt stuff *
* *
******************************************************************/
#ifdef ENABLE_INTERRUPT_CODE
/* This is the interrupt handler for the 21554. It currently does
* nothing useful; it just prints out a message and clears the
* interrupt */
static void ebsamem_interrupt(int irq,
void *data,
struct pt_regs *regs)
{
  int ret;
  PDEBUG(1, "Got interrupt");

  /* The interrupt must be cleared or the system will hang */
  ret = iClearCoHostINT(g_pcidev);
  if (ret < 0) {
    printk(KERN_ERR "ebsamem: Error clearing interrupt\n");
  } else if (ret == 0) {
    PDEBUG(1, "Attempted to clear nonexistent interrupt");
  } else if (ret == 1) {
    PDEBUG(1, "Cleared interrupt");
  } else {
    printk(KERN_ERR "ebsamem: Unknown return value from \n    iClearCoHostINT\n");
  }
}
#endif

/******************************************************************
* * Shared memory access for kernel *
* *
******************************************************************/
/* Provides access to the shared memory by other kernel code.
* Parameters:
*   addr (OUT): Kernel Virtual Address of the shared memory
*   size (OUT): Size of the shared memory
*/
void get_ebsa_shrmem(unsigned long *addr, unsigned long *size)
{
  if (!addr || !size) {

printk(KERN_ERR "get_ebsa_shrmem: invalid argument\n");
return;
}

/* Prevent module unload while shared memory is in use */
MOD_INC_USE_COUNT;

*addr = (unsigned long)shm_addr;
*size = (unsigned long)SHRMEM_SIZE;
}

void release_ebsa_shrmem(void)
{
MOD_DEC_USE_COUNT;
}

/******************************************************************
* *
* Init functions and module functions *
* *
******************************************************************/

/* This function sets up the 21285 and 21554 for the shared memory *
* segment. It is called automatically by the PCI device manager. *
* Parameters: *
* pcidev (IN): PCI device struct for 21554 *
* pciid (IN): PCI device id for 21554 (unused) */
static int __devinit ebsapoll_probe(struct pci_dev *pcidev,
struct pci_device_id *pciid)
{
unsigned long phys_addr;
int rc;

if (!pcidev || !pciid)
return -EFAULT;

/* Get the physical location of our SDRAM window */
phys_addr = (unsigned long)virt_to_phys(shm_addr);

/* This gets the PCI address where the shared memory is mapped. *
* __virt_to_bus() returns the sum of the PCI address of the *
* SDRAM window and the offset of the argument from the start *
* of SDRAM */
ebsa_pci_addr = __virt_to_bus((unsigned long)shm_addr);
printk(KERN_INFO "EBSA PCI addr is %#lx\n", ebsa_pci_addr);

#ifdef ENABLE_INTERRUPT_CODE
/* Obtain IRQ number and the unique device ID from 21554 */
if (iGet_irq_info(pcidev, &irq, &dev_id) < 0) {
    printk(KERN_ERR "ebsamem: Failed to get IRQ info\n");
    return -ENODEV;
} else {
    PDEBUG(1, "IRQ %d found", irq);
}

/* Tell Linux about our interrupt handler */
if (request_irq(irq,
ebsamem_interrupt,
SA_SHIRQ,
"ebsamem",
&dev_id)) {
    printk(KERN_ERR "ebsamem: Couldn't get interrupt %d\n", irq);
    return -ENODEV
} else {
    PDEBUG(1, "Assigned IRQ %d", irq);
    got_irq = 1;
}
#endif
/* Configure the 21554 with the address */
rc = i21554_cohost_config(pcidev, ebsa_pci_addr);
if (rc < 0) {
    printk(KERN_ERR "ebsamem: 21554 configuration failed \n (rc=%d)\n", rc);
    return rc;
}
/* Save the pci_dev struct passed to this function */
g_pcidev = pcidev;
/* Assign a /dev node */
rc = devfs_register_chrdev(EBSA_MAJOR, "ebsamem", &ebsa_fops);
if (rc < 0) {
    printk(KERN_ERR "ebsamem: can't get major %d (rc=%d)\n",
        EBSA_MAJOR,
        rc);
    return rc;
}
return 0;
}
/* This is called when either the device is removed or the module
 * is unloaded. It removes the device node and attempts to disable
 * the 21554 */
static void __devinit ebsapoll_remove(struct pci_dev *pcidev)
{
    int rc;

    if (devfs_unregister_chrdev(EBSA_MAJOR, "ebsamem") < 0)
        printk(KERN_ERR "ebsamem: can't unregister dev on major %d\n",
            EBSA_MAJOR);
#ifdef ENABLE_INTERRUPT_CODE
    if (got_irq)
        free_irq(irq, &dev_id);
#endif
    /* Disable 21554 */
    rc = i21554_cohost_config(pcidev, NULL);
    if (rc < 0) {
        printk(KERN_ERR "ebsamem: couldn't disable 21554 (rc=%d)\n",
            rc);
        return rc;
    }
The pci_dev structure is no longer valid

g_pcidev = NULL;

This lists the PCI IDs that will trigger the ebsamem_probe function

static struct pci_device_id id_table_21554[] __devinitdata = {
    { VENDOR_21554, DEV_ID_21554, PCI_ANY_ID, PCI_ANY_ID, 0, 0 },
    { 0, }
};

MODULE_DEVICE_TABLE(pci, id_table_21554);

static struct pci_driver ebsa_driver = {
    name: "ebsamem",
    id_table: id_table_21554,
    probe: ebsapoll_probe,
    remove: ebsapoll_remove
};

Main entry point. Called when kernel boots (or if built as a module, by init_module() upon module load.)

static int __init ebsapoll_init(void)
{
    int rc;

    printk(KERN_INFO "EBSA shared mem driver version %s\n", version);

    if (!pci_present())
        return -ENODEV; /* No PCI bus in this machine! */

    /* Initialize the open lock to an unlocked state */
    init_MUTEX(&open_lock);

    /* Allocate the shared memory */
    rc = alloc_mem();
    if (rc < 0) {
        printk(KERN_ERR "ebsapoll: Memory init failed\n");
        goto out;
    }

    /* Register ourselves with the PCI subsystem */
    rc = pci_module_init(&ebsa_driver);
    if (rc < 0) {
        printk(KERN_ERR "ebsapoll: PCI init failed\n");
        goto out_dealloc;
    }

    return 0;

out_dealloc:
    free_mem();
out:
    return rc;

/* Clean up, in preparation for module unload */
static void __exit ebsapoll_exit(void)
{
    printk(KERN_INFO "EBSA shared memory driver exiting\n");
    pci_unregister_driver(&ebsa_driver);
    free_mem();
}

/* Module entry/exit points */
#ifdef MODULE
    int init_module(void)
    {
        return ebsapoll_init();
    }

    void cleanup_module(void)
    {
        ebsapoll_exit();
    }
#endif

    /* Export the shared memory access functions to other modules */
    EXPORT_SYMBOL(get_ebsa_shrmem);
    EXPORT_SYMBOL(release_ebsa_shrmem);
#include <linux/kernel.h>
/* Debug level. 1 is least, 5 is greatest. Undefine to disable. */
#define HOSTMEM_DEBUG 1

#define CINIC_IOC_INTERRUPT _IOW (0xAA, 128, u16)

#define HOST_MAJOR 253
#define HOST_MINOR 0

#include <linux/hostmem.h>


```c
#include <linux/kernel.h>
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/init.h>
#include <linux/ctype.h>
#include <linux/smp_lock.h>
#include <linux/fs.h>
#include <linux/vmalloc.h>
#include <linux/slab.h>
#include <linux/pagemap.h>
#include <linux/pci.h>
#include <asm/semaphore.h>
#include <linux/ioport.h>
#include <linux/wrapper.h>
#include <linux/devfs_fs_kernel.h>
#include <asm/uaccess.h>

#include "global.h"
#include "hostmem.h"
#include "21554.h"

#define ENABLE_INTERRUPT_CODE

MODULE_AUTHOR("Mark McClelland (3Com/Cal Poly Project)\n
MODULE_DESCRIPTION("Host shared memory driver\n
/***************************************************************************/
/* Global variables */
***************************************************************************/

// Driver version string */
static const char version[] = "1.03";

/* IRQ number of 21554 */
static unsigned int irq = -1;

/* 21554 PCI address */
static void *pci_addr;

/* Unique identifier for our particular 21554 */
static u32 dev_id;

/* Kernel Virtual Address of shared memory */
static void *shm_addr;
```
/* Prevents multiple simultaneous open()'s */
static struct semaphore open_lock;

/* Indicates that IRQ was successfully registered */
static int got_irq;

/* The 21554 PCI device struct */
static struct pci_dev *g_pcidev;

/******************************************************************
* *
* System calls - These provide user-space access to the shared memory when needed.
* *
*******************************************************************/

static int hostmem_open(struct inode *inode, struct file *file)
{
    /* Only allow one process at a time */
    down(&open_lock);

    printk(KERN_INFO "hostmem: open\n");

    /* Prevent module from being unloaded while open */
    MOD_INC_USE_COUNT;

    return 0;
}

static int hostmem_close(struct inode *inode, struct file *file)
{
    printk(KERN_INFO "hostmem: close\n");

    /* Allow module to be unloaded */
    MOD_DEC_USE_COUNT;

    /* Allow the next open() to succeed */
    up(&open_lock);

    return 0;
}

/*
* This provides the user a window onto the shared memory buffer.
* Code partially taken from linux/drivers/usb/ov511.c by Mark McClelland and from "Linux Device Drivers" by Alessandro Rubini*/

static int hostmem_mmap(struct file *file, struct vm_area_struct *vma)
{
    /* Starting physical address of SDRAM region to map */
    unsigned long phys_start;

    /* Requested size of mapping */
    unsigned long size = vma->vm_end - vma->vm_start;

    /* Requested offset of mapped area from beginning of shared
unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

if (pci_addr == NULL)
    return -ENOMEM;

/* Did user ask for too much? (Too little is OK) */
if (size > SHRMEM_SIZE)
    return -EINVAL;

/* Give them a warning if they asked for too little, though */
if (size != SHRMEM_SIZE)
    PDEBUG(1, "warning: requested size does not match \n    SHRMEM_SIZE\n");

/* Calculate the starting address, based on the offset */
phys_start = (unsigned long) pci_addr + offset;

PDEBUG(1, "remap_page_range(0x%lx, 0x%lx, %ld, ...)
    vma->vm_start, virt_to_phys((void*)phys_start), size);

/* Do the page mapping */
if (remap_page_range(vma->vm_start, phys_start, size, vma->vm_page_prot)) {
    printk(KERN_ERR "remap_page_range() failed\n");
    return -EAGAIN;
}

return 0;

static int hostmem_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
    switch(cmd) {
    case CINIC_IOC_INTERRUPT:
    {
        #ifdef ENABLE_INTERRUPT_CODE
        u16 doorbell_val;

        /* Copy the data from user-space */
        if (copy_from_user(&doorbell_val, (void *)arg, sizeof(doorbell_val)))
            return -EFAULT;

        PDEBUG(1, "Sending interrupt to ebsa (doorbell=%d)",
            doorbell_val);

        /* Send the interrupt */
        if (iInterrupt_cohost(g_pcidev, doorbell_val))
            return -EIO;
        #endif
        return 0;
    }
}
/* System call table */
struct file_operations hostmem_fops = {
    open: hostmem_open,
    release: hostmem_close,
    mmap: hostmem_mmap,
    ioctl: hostmem_ioctl,
};

/********************************************************************************
* Interrupt stuff
********************************************************************************

#ifndef ENABLE_INTERRUPT_CODE

/* This is the interrupt handler for the 21554. It currently does nothing useful; it just prints out a message and clears the interrupt */
static void hostmem_interrupt(int irq, void *data, struct pt_regs *regs)
{
    int ret;

    PDEBUG(1, "Got interrupt");

    /* The interrupt must be cleared or the system will hang */
    ret = iClearHostINT(g_pcidev);
    if (ret < 0) {
        printk(KERN_ERR "hostmem: Error clearing interrupt\n");
    } else if (ret == 0) {
        PDEBUG(1, "Attempted to clear nonexistent interrupt");
    } else if (ret == 1) {
        PDEBUG(1, "Cleared interrupt");
    } else {
        printk(KERN_ERR "hostmem: Unknown return value from iClearHostINT\n");
    }
}
#endif
/* Provides access to the shared memory by other kernel code.
 * Parameters:
 *   addr (OUT): Kernel Virtual Address of the shared memory
 *   size (OUT): Size of the shared memory
 */
void get_host_shrmem(unsigned long *addr, unsigned long *size)
{
    if (!addr || !size) {
        printk(KERN_ERR "get_host_shrmem: invalid argument\n");
        return;
    }

    /* Prevent module unload while shared memory is in use */
    MOD_INC_USE_COUNT;

    *addr = (unsigned long)shm_addr;
    *size = (unsigned long)SHRMEM_SIZE;
}

void release_host_shrmem(void)
{
    MOD_DEC_USE_COUNT;
}

/******************************************************************
 * Init functions and module functions *
 ******************************************************************/
/* This function sets up the 21554 for the shared memory
 * segment. It is called automatically by the PCI device manager.
 * Parameters:
 *   pcidev (IN): PCI device struct for 21554
 *   pciid (IN): PCI device id for 21554 (unused)
 */
static int __devinit host_21554_probe(struct pci_dev *pcidev,
    struct pci_device_id *pciid)
{
    int rc;

    if (!pcidev || !pciid)
        return -EFAULT;

#ifdef ENABLE_INTERRUPT_CODE
{
    /* Obtain IRQ number and the unique device ID from 21554 */
    if (iGet_irq_info(pcidev, &irq, &dev_id) < 0) {
        printk(KERN_ERR "hostmem: Failed to get IRQ info\n");
        return -ENODEV;
    } else {
        PDEBUG(1, "IRQ %d found", irq);
    }

    /* Tell Linux about our interrupt handler */
    if (request_irq(irq, hostmem_interrupt,}
SA_SHIRQ,
"hostmem",
&dev_id}) {
    printk(KERN_ERR "hostmem: Could not get interrupt %d\n",
            irq);
    return -ENODEV
} else {    
    PDEBUG(1, "Assigned IRQ %d", irq);
    got_irq = 1;
}
#endif

/* Get the physical location of our SDRAM window */
pci_addr = (void *) ui21554_host_config(pcidev);

/* Create a kernel virtual mapping of it */
shm_addr = (void *) ioremap((unsigned long)pci_addr, SHRMEM_SIZE);

printk(KERN_INFO "21554 is at %#lx/%#lx\n",
        (unsigned long)shm_addr, (unsigned long)pci_addr);

/* Save the pci_dev struct passed to this function */
g_pcidev = pcidev;

/* Assign a /dev node for it */
rc = devfs_register_chrdev(HOST_MAJOR, "hostmem",
                           &hostmem_fops);
if (rc < 0) {
    printk(KERN_ERR "hostdrv: can't get major %d\n",
            HOST_MAJOR);
    return rc;
}

return 0;
}

/* This is called when either the device is removed or the module
 * is unloaded. It removes the device node and attempts to disable
 * the 21554 */
static void __devinit host_21554_remove(struct pci_dev *pcidev)
{
    if (devfs_unregister_chrdev(HOST_MAJOR, "hostmem") < 0) {
        printk(KERN_ERR "hostmem: can't unregister dev on major \
                %d\n", HOST_MAJOR);
    }

    /* Disable 21554 */
    rc = i21554_host_config(pcidev, NULL);
    if (rc < 0) {
        printk(KERN_ERR "hostmem: couldn't disable 21554 (rc=%d)\n",
                rc);
    }
    return rc;
#ifdef ENABLE_INTERRUPT_CODE
if (got_irq)
    free_irq(irq, &dev_id);
#endif

vfree(shm_addr);
shm_addr = NULL;
pci_addr = NULL;
g_pcidev = NULL;
}

static struct pci_device_id id_table_21554[] __devinitdata = {
    { VENDOR_21554, DEV_ID_21554, PCI_ANY_ID, PCI_ANY_ID, 0, 0 },
    { 0, }
};
MODULE_DEVICE_TABLE(pci, id_table_21554);

static struct pci_driver hostmem_driver = {
    name: "hostmem",
    id_table: id_table_21554,
    probe: host_21554_probe,
    remove: host_21554_remove
};

/* Main entry point. Called when kernel boots (or if built as a
 * module, by init_module() upon module load.) */
static int __init hostmem_init(void)
{
    int rc;

    printk(KERN_INFO "Host shared memory driver (hostmem) version \n \
 version\n",
           version);

    if (!pci_present())
        return -ENODEV; /* No PCI bus in this machine! */

    /* Initialize the open lock to an unlocked state */
    init_MUTEX(&open_lock);

    /* Register ourselves with the PCI subsystem */
    rc = pci_module_init(&hostmem_driver);
    if (rc < 0)
        printk(KERN_ERR "hostmem: PCI init failed\n");

    return 0;
}

static void __exit hostmem_exit(void)
{
    printk(KERN_INFO "host shared memory driver (hostdrv) exiting\n");

    pci_unregister_driver(&hostmem_driver);
}
/* Module entry/exit points */
#ifdef MODULE
int init_module(void)
{
    return hostmem_init();
}

void cleanup_module(void)
{
    hostmem_exit();
}
#endif

/* Export the shared memory access functions to other modules */
EXPORT_SYMBOL(get_host_shrmem);
EXPORT_SYMBOL(release_host_shrmem);
testapp_ebsa.c

/**************************************************************************
* Test application for ebsamem driver
*
* Does the following:
* 1. Creates a user-space mapping of the shared memory
* 2. Displays bytes 2 and 3 of the shared memory
* 3. Modifies bytes 0 and 1 of the shared memory
*
* Author: Mark McClelland (3Com/Cal Poly Project)
* Date: 12/6/2000
**************************************************************************/

#include <stdio.h>
#include <stddef.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "common.h"

int main(void)
{
    int fd, rc;
    unsigned char * mem;
    unsigned long memsize = SHM_SIZE;

    /* Open the device node */
    fd = open("/dev/ebsamem", O_RDWR);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    /* Create the mapping. It is created with the MAP_SHARED
     * option so that the changes are permanent stored in the
     * shared memory rather than in a local copy */
    mem = mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (mem == NULL) {
        perror("mmap (1)"_COMPONENT_1);
        exit(1);
    }

    printf("mem[2] = %d, mem[3] = %d\nShould be 13 and 37\n", mem[2], mem[3]);

    /* Modify the memory */
    mem[0] = 95;
    mem[1] = 46;

    /* Close the device node */
close(fd);

/* Destroy the shared memory mapping */
rc = munmap(mem, SHM_SIZE);
if (rc) {
    perror("munmap");
    exit(1);
}

return 0;
**testapp_host.c**

/**************************************************************************
* Test application for hostmem driver
* 
* Does the following:
* 1. Creates a user-space mapping of the shared memory
* 2. Displays bytes 0 and 1 of the shared memory
* 3. Modifies bytes 2 and 3 of the shared memory
* 4. Unmaps the shared memory
* 
* Author: Mark McClelland (3Com/Cal Poly Project)
* Date: 12/6/2000
**************************************************************************/

#include <stdio.h>
#include <stddef.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "common.h"

int main(void)
{
    int fd, rc;
    unsigned char * mem;
    unsigned long memsize = SHM_SIZE;

    /* Open the device node */
    fd = open("/dev/hostmem", O_RDWR);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    /* Create the mapping. It is created with the MAP_SHARED */
    /* option so that the changes are permanent stored in the */
    /* shared memory rather than in a local copy */
    mem = mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (mem == NULL) {
        perror("mmap (1)");
        exit(1);
    }

    printf("mem[0] = %d, mem[1] = %d\nShould be 95 and 46\n",
        mem[0],
        mem[1]);

    /* Modify the memory */
    mem[2] = 13;
    mem[3] = 37;

    /* Close the device node */
close(fd);

/* Destroy the shared memory mapping */
rc = munmap(mem, SHM_SIZE);
if (rc) {
    perror("munmap");
    exit(1);
}
return 0;
int_test.c

/**************************************************************************
 * Test application for hostmem/ebsamem interrupt capability
 * Initiates an interrupt from the hostmem or ebsamem driver
 * Usage: int_test <device_node> <16_bit_val>
 * Where <device_node> is the name of the /dev node exported by the
driver
 * <16_bit_val> is an arbitrary number to be written into the
doorbell register. Must be > 0.
 * Author: Mark McClelland (3Com/Cal Poly Project)
 * Date: 5/18/2000
***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>
#include <sys/ioctl.h>
/* Definition of ioctl: Send an interrupt */
#define CINIC_IOC_INTERRUPT _IOW (0xAA, 128, uint16_t)

int main(int argc, char * argv[]) {
    int fd;
    long int int_arg;
    uint16_t doorbell_val;

    /* Check that user supplied two arguments */
    if (argc != 3) {
        fprintf(stderr, "You must supply the name of the device node \n and a 16-bit value\n");
        return 1;
    }

    /* Convert string to integer. 0 as third param means allow any
     * number base */
    int_arg = strtol(argv[2], NULL, 0);
    if (int_arg < 0 || int_arg > 65535) {
        fprintf(stderr, "Error: integer value (argument 2) out of \n range\n");
        return 1;
    }

    /* Cast it to unsigned 16 bit */
    doorbell_val = (uint16_t)int_arg;

    /* Open the device node */
    fd = open(argv[1], O_RDWR);
    if (fd == -1) {
        perror("open");
        return 1;
    }
/* Send the interrupt */
if (ioctl(fd, CINIC_IOC_INTERRUPT, &doorbell_val) == -1) {
    perror("CINIC_IOC_INTERRUPT");
    return 1;
}

printf("Sent interrupt with doorbell value %d\n", doorbell_val);
close(fd);
return 0;
Appendix C - The Source Code CD

A CD is available that contains the following source code:

All of my code for this project

Jason Hatashita's 21554.c and 21554.h files

Rob McCready's network protocol code

The Debian FTP test application

Makefile for all of the code

The Big Physical Area Patch

The CD may be found in the CiNIC lab at Cal Poly. You may also e-mail me at mark@alpha.dyndns.org to request a copy.