ARM Linux on the Intel *Assabet* Development Board

John G Dorsey
<john+@cs.cmu.edu>

Technical Report
ICES #0x-yy-00

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This report describes the contributions of the Wearable Group at Carnegie Mellon University to the ARM architecture port of the Linux kernel. Several fundamental ARM Linux concepts are presented in overview, including kernel loading and initialization. A number of feature introductions are detailed which chronicle the history of Linux on the Intel SA-1110 Development Board, *Assabet*, as well as the SA-1111 Microprocessor Companion Chip Development Board, *Neponset*. Covered work includes the topics of memory configuration, serial I/O, the SA-1111, and PCMCIA support. The appendices describe the organization of the ARM Linux patch system, and catalog the patches released by the Wearable Group which support *Assabet*.

1 Introduction

The Linux operating system[Lin], in a preliminary release of version 2.4 at the time of this writing, is available on at least twelve separate processor architectures, including the Advanced RISC Machines[ARMa] family. The StrongARM processor, a high-performance, low-power part based around the SA-1 (similar to the ARM8) core, is supported by a separate architecture sub-port within the ARM Linux development project[ARMb]. Prior to the development efforts discussed in this report, Linux was available on the SA-110 and SA-1100 processors in a variety of systems, such as the *Brutus* development board and the Compaq (*née* Digital Equipment Corporation) *Itsy*[Itsa]. This report describes the steps taken by the ARM Linux community to support the SA-1110[SA1a] and SA-1111[SA1b] in the kernel, and focuses in particular on the contributions of the Wearable Group[Wea] at Carnegie Mellon University.

2 Overview of ARM Linux

As a Unix derivative, the Linux operating system is organized into a kernel and a number of user-level programs. It has been the experience of the author that a large number of userland programs which are already known to work under Linux on some other architecture (say, `i386` or `ppc`) can be ported to the ARM with relatively little effort.[1] As such, most of

*This work was funded by National Science Foundation contract EIA-9901321-NSF, DARPA contract N660019928918-N306A, and Intel Corporation.

[1] This is to say, such programs can be made to be functionally correct generally just by building them with the appropri-

the material in this report will concern itself with the kernel. Having said this, it is worth noting that none of the ARM Linux kernel work could take place without the efforts of Philip Blundell on the ARM ports of the GNU C library, GNU *binutils*, and GNU Compiler Collection (*gcc*). These resources presently support version 4 of the ARM architecture, which the SA-1110 implements.

The Linux kernel was not originally intended to support architectures beyond the 80386 found in the personal computer of its creator. Although a number of steps to correct this have been taken, evidence of the `i386` bias remains easy to find in the current release. For example, many device drivers assume the existence of an I/O address space, and dedicated I/O instructions through which to access this region. Nevertheless, many key kernel services are now sufficiently architecture-independent as to enable new ports — such as the SA-1110 work described herein — to proceed in a straightforward manner.

In order to present the portions of the kernel which have relevance to a discussion of the SA-1110 port, the following subsections describe the bootstrap process on a generic StrongARM SA-1100 or SA-1110 system. Beginning with the boot loader, we review the low-level hardware initialization steps which are required for the kernel to begin execution. Next discussed is a feature of Linux which allows the kernel to boot from a compressed program image. Following this will be a summary of kernel memory initialization and low-level resource setup, such as that for interrupts. The overview concludes with an explanation of a typical method for launching userland.

2.1 The Boot Loader

The StrongARM SA-1100 microprocessor, upon emerging from reset, begins executing code from the base of static memory bank 0, which is generally mapped to some quantity of non-volatile flash RAM.[2] Because of this convention, a design will generally involve a program, stored in bank 0, which maintains responsibility for certain hardware diagnostics, communication with a workstation development environment, and the task of populating physical memory with kernel images and other data. This program, referred to as the "boot loader" in this document,[3] changes infrequently, and if corrupted, generally renders the system unusable.

The StrongARM SA-1100 processor includes features which allow it to operate at a number of discrete multiples of a core clock frequency. Because this has implications for the memory access times presented to the available RAM components, one of the first responsibilities of a boot loader is to properly configure the processor clock and memory access settings. Once the RAMs are accessible, programmatic constructs such as a heap or stack can be initialized, if so desired.[4]

Following memory initialization, the boot loader may enable interrupts from the processor serial port, which will allow communication with the development environment. Multiple serial ports are available on the StrongARM SA-1100, and it is at the discretion of the boot loader developer to assign a role to each one. For example, the boot loader might accept command and data traffic on one UART, while displaying debugging information over another. The choice of which port to configure, and the selection of parameters used (*e.g.*, data rate, parity), must be consistent with the assumptions made in the kernel startup code.[5] Once a serial connection is available, the boot loader may accept commands from a development host. One type of command might instruct the boot loader to accept a sequence of data over the

ate toolchain. One general exception involves programs which employ large amounts of floating point computation; as most ARM implementations lack floating point hardware, these operations are emulated with integer arithmetic, at a noticeable performance penalty.

[2]As a matter of trivia, this phrasing is chosen to carefully avoid the suggestion that the memory is directly addressed through bank 0. The *Itsy* in fact contains flash memory only in banks 1 (the motherboard) and 2 (the optional daughtercard), and relies on a user setting to determine which of these become mapped to "bank 0."

[3]Other names for this program include "firmware," "boot monitor," or simply, "monitor."

[4]Any such structures written to memory will be lost when the kernel begins executing. This restricts the amount of communication which can occur between the boot loader and the kernel to the contents of the processor registers.

[5]Otherwise, any console information displayed by the kernel during startup will be unintelligible when viewed from a terminal configured for the boot loader settings.

serial connection and write that data to a location in memory, such as RAM or flash RAM. Another command might direct the boot loader to perform a series of hardware diagnostics, such as a memory integrity test. A third command might direct the processor to reset and begin executing instructions at a specified location in memory.

Not all systems will rely on an attached host for operation. An independent system may boot by copying a program image from non-volatile storage (such as flash RAM or a rotating disk) into RAM and jumping to that program. A system may also initialize a network interface, such as a wireless LAN adapter, and request a program image from a network file server.

Regardless of how the kernel image is obtained, ARM Linux does impose three requirements on the boot loader which must be satisfied before the kernel may be entered. The first of these is that the memory management unit must be disabled, register `r0` must contain the value `0x0`, and register `r1` must contain a unique architecture identifier.[6] No other information is communicated to the kernel beyond these two values. Once these preconditions have been met, the boot loader is free to perform a jump to the beginning of the kernel image, and begin executing instructions.

2.2 The Compressed Kernel Image

The Linux kernel, configured in a manner typical for a StrongARM host, is now over six megabytes in size[7]. For kernel developers, this is unhappy news, as a complete kernel image must be moved over a serial link — at 115200bps — each time a code change is made. Adding an eight minute delay to the compile–execute–debug cycle seems unattractive, but worse, some systems lack sufficient non-volatile storage to house such a large kernel image.[8]

Fortunately, the kernel program text is highly compressible — a size reduction by a factor of ten is not uncommon — and so a solution to both the delay and storage problems just mentioned seems to have presented itself. Unfortunately, as feature-filled as the StrongARM may be, the ability to execute compressed machine instructions has not yet been included. One could imagine placing decompression features in the boot loader, but not all loaders have been developed with Linux in mind.[9] A third solution is to piggyback a small decompression program onto the compressed kernel image.

From the perspective of the boot loader, executing a kernel compressed in this manner, known as a *zImage*, is no different from executing a "normal" program image. The processor jumps to the decompressor, which reads the compressed image from memory, expands the image (using, for example, the Lempel-Ziv algorithm of *gzip*), and places the resulting instructions and data in physical memory. The decompressor then jumps to the kernel entry point, and kernel startup begins as usual.

2.3 Kernel Initialization

Once the kernel program image is in memory — regardless of how it got there — execution begins with a small amount of code written in assembly which initializes the system. The kernel begins by attempting to discover on what type of machine it is executing, first by checking the processor ID register, then by examining the unique architecture identifier passed in register `r1`. If the system is recognized, the kernel constructs the processor page tables, which include direct mappings for the first few megabytes of the kernel image. Any data in the BSS segment are zeroed, the stack pointer is initialized, and the processor branches to the high-level kernel startup routine, which is written in C.

At this point, the kernel begins to allocate various resources such as memory maps and interrupts. Different system designs may have varying quantities of DRAM, SDRAM, or flash RAM installed at varying locations in the system address space. Further, I/O devices — either external to the processor (such as a coprocessor), or internal (such as the Peripheral Control Module) — must have virtual-to-physical map-

[6] For example, *Assabet* has been assigned the value `0x19`.

[7] Strictly speaking, the ELF image is six megabytes; this version can be stripped to produce a 1.3MB image.

[8] For example, the *Itsy*, *sans* daughtercard, contains 4MB of flash RAM.

[9] *Angel*, the reference ARM boot loader, was for nearly a year the only program which could boot *Assabet*. It does not provide decompression support.

pings in order to be accessible. The kernel selects memory and I/O mapping lists to add to the page table based on knowledge of the system type, which was passed by the boot loader as the architecture identifier. (I/O mappings can also be created or destroyed dynamically.)

In order to describe how interrupts are handled under Linux running on the StrongARM SA-1100, a brief discussion of StrongARM SA-1100 interrupt sources seems in order. There are 32 separate interrupts which can be recognized by the processor, which arrive from sources such as the operating system timers, serial ports, and general-purpose I/O (GPIO) pins. Only twelve interrupts are allocated to the 28 GPIO sources, which leads to a potentially confusing interrupt handling strategy. Interrupts 0–10 map directly to edge detects on GPIO pins 0–10. Interrupt 11 absorbs all of GPIO pins 11–27. Any interrupt handling implementation would have to receive IRQ 11, then consult the GPIO edge detect register and manually deal with the multiplexed GPIO sources. The original method of handling these interrupts was for device drivers to request IRQ 11 in shared mode. At runtime, each driver which had requested this interrupt would be entered, given the chance to examine the GPIO edge detect register, and possibly handle a pending device interrupt. As will be discussed later, this approach has subsequently been replaced by a somewhat easier-to-use mechanism.

At this point, the kernel can proceed with a number of architecture-independent initializations, such as setting up the console, enabling filesystems and caching, and launching the *init* thread. This thread begins by performing a number of setup tasks which require the processor, memory, and other basic resources to be initialized. One operation performed by this thread is to mount the root filesystem, which on many StrongARM systems involves the use of a RAM disk.

A RAM disk is a compressed filesystem image which can be loaded into memory,[10] and then mounted just like any other volume, such as a disk. RAM disks tend to be small; a compressed disk image might be 1.5–2.5MB in size. Such a disk might contain the familiar `/etc` initialization scripts, userland utilities such as interactive shells and editors, and loadable kernel modules. A filesystem mounted from a RAM disk may be writeable, but in general, any modifications to a RAM disk filesystem will not survive a reset or loss of power. The Journalling Flash File System, which would allow modifications to propagate through to flash RAM, is being developed by Axis Communications.

The last task performed by the *init* thread during startup is to *exec()* a program from the root filesystem which will be responsible for bringing up userland. In this document, “userland” will refer to the set of programs and data which exist outside of the kernel address space and which do not have the ability to execute privileged instructions. Informally, this will mean the set of programs and data with which the user can interact, such as shells or editors. The program run by the *init* thread[11] usually begins by executing a number of shell scripts which activate system services such as *inetd* or *syslogd*. An *init* program may also be responsible for running a program that handles login terminals, such as *getty*. Note that *init* is the parent of all processes that execute on an ARM Linux system, and as such, is indirectly responsible for all user interaction with the system, regardless of whether this occurs via a terminal, a graphics console, voice, or some other mode.

3 The Port to *Assabet*

Due to strong similarities between the SA-1100 and SA-1110 microprocessors, a great deal of usable code was already available in ARM Linux by the time *Assabet*[Ass] was released. The initial porting activity, which was undertaken by several parallel groups, involved only a few changes to the memory setup

[10] The initial RAM disk containing the root filesystem will generally be loaded by the boot loader at the same time as the kernel image is placed in memory. This RAM disk can come from flash memory, a serial connection to a development host, or any other source available to the boot loader.

[11] The possibilities are, in order, a user-specified program passed as a kernel parameter, `/sbin/init`, `/etc/init`, `/bin/init`, and `/bin/sh`. If none of these are available, a kernel panic is triggered.

and serial port configuration. The initial *Assabet* patch release by the Wearable Group occurred on 13 April, 2000, for kernel version 2.3.99.[12]

In subsequent months, a number of additional features have been added to the kernel which improve support for *Assabet*. The *Neponset*[Nep] board is now operational, with its SA-1111 companion processor. PCMCIA[PCM] implementations for both the *Assabet* CompactFlash[Com] slot and the *Neponset* PCMCIA and CF slots are available. Drivers for a CompactFlash ethernet device, IDE storage devices such as the IBM Microdrive[Mic], and the Lucent WaveLAN/IEEE 802.11[Wav] interface are all functional.

The next section describes the specific contributions to ARM Linux made by the Wearable Group to improve *Assabet* support in the kernel. Following this is a section describing work on ARM Linux performed by other groups external to Carnegie Mellon involving *Assabet*.

3.1 Work at Carnegie Mellon

Development on *Assabet* at Carnegie Mellon follows more than a year of work on another StrongARM-based platform, *Itsy*. In addition to low-power research on the standalone *Itsy*, which was developed at the Digital Equipment Corporation Western Research Laboratory (DEC WRL, now Compaq WRL), more recent work has centered around an enhanced version of the *Itsy* which includes a higher-capacity lithium ion battery and dual PCMCIA slots. This platform, known as the *Itsy/Cue*[Itsb], has been used for research in the areas of audio interfaces and mobile wireless networking.

The introduction of the PCMCIA slots to the *Itsy* hardware presented an interesting challenge in terms of software support. The official kernel release for *Itsy* at the time of this work was a derivative of Linux 2.0.30 which was somewhat out of sync with the standard ARM Linux tree. A standalone software package, PCMCIA Card Services for Linux, was available for architectures which used one of a small number of PCMCIA controllers, such as the Intel 82365. In order to support the built-in PCMCIA control functionality of the SA-1100, an effort was undertaken by the Wearable Group to adapt Card Services to the StrongARM. The first patch to Card Services 3.1.8 was released on 16 January, 2000.

Figure 1: *Itsy/Cue* with PCMCIA support.

This early work on ARM Linux and PCMCIA Card Services served as a foundation for development of subsequent patches involving *Assabet*. A detailed explanation of the Card Services implementation for ARM Linux appears later in this section. Also covered in this section are the topics of kernel configuration, memory, serial ports, debug LEDs, *Neponset*, the SA-1111, serial audio control, and PCMCIA client drivers.

3.1.1 Kernel Configuration

The original *Assabet* patch released by the Wearable Group contained a number of changes to the naming conventions used within the kernel source tree. The changes generalized the nomenclature from "SA-1100" to "SA-11x0," the better to capture the fact that much of the code applies equally to the SA-1100 and SA-1110 processors. The ARM Linux community chose not to adopt this change, preferring instead to follow the model of other architectures supported by Linux. (For example, `i386` encapsulates the 80486 and various models of the Pentium processor.)

In the current ARM Linux kernel, the symbol

[12] Specifically, *2.3.99-pre3-rmk1-np3*. See the appendices for information on the ARM Linux patch system.

CONFIG_ARCH_SA1100 is defined whenever the kernel is configured for SA-1100 or SA-1110 processors. Support for *Assabet* is indicated by the symbol CONFIG_SA1100_ASSABET, and additional support for *Neponset* is indicated by CONFIG_ASSABET_NEPONSET. Supporting *Neponset* implies support for the SA-1111 companion processor, which is indicated by the symbol CONFIG_SA1111.

3.1.2 Memory

Assabet includes a quantity[13] of 32-bit SDRAM in dynamic RAM bank 0. *Neponset*, when present, provides additional memory running at half the speed of the *Assabet* SDRAM. During startup, when the kernel detects that it is running on *Assabet*, an entry for the single bank of SDRAM at 0xc0000000 is added to the database of memory resources. Space for an initial RAM disk, up to 3MB[14] in size, is reserved at 0xc0800000. Kernel — or *zImage* — execution begins at 0xc0008000, with the page tables residing below this address. The final RAM disk and kernel image locations in SDRAM were selected by Nicolas Pitre.

In addition to RAM, the StrongARM SA-1100 memory map contains regions of the address space which allow access to I/O devices. In order to operate on these regions, a virtual mapping for the physical address ranges must be constructed in the kernel page tables. Such mappings can be constructed on demand, but it is convenient to install several mappings at startup time. These mappings include space for the Peripheral Control Module (USB, serial I/O), the System Control Module (interrupts, GPIO), Memory Control (RAM configuration, PCMCIA timing), LCD control, DMA control, and access to the I/O, attribute, and common memory spaces of both PCMCIA slots. Of course, these regions are common to all SA-1100 and SA-1110 systems; when the kernel is running on *Assabet*, several additional ranges are mapped as well. These include the *Assabet* Board Control Register, the *Neponset* CPLD registers, and the SA-1111 itself.

3.1.3 Serial I/O

As mentioned previously, a boot loader may obtain kernel and filesystem images at boot time via a serial port on the processor. Once control has been transferred to the kernel, however, serial I/O continues to be important as it may be the primary method of interaction with the system. *Assabet* provides one RS-232 interface, routed to serial port 1 on the SA-1110. *Neponset* provides additional hardware for two RS-232 ports, one of which is routed to serial port 3, the other preempts the interface on *Assabet* to implement port 1.

Because of a design choice in the development of the *Angel* boot loader, which is the most common boot loader used with *Assabet*, the rôles of the various serial ports are somewhat muddled. When *Assabet* is booted without the companion board, all of the *Angel* debug messages are of course directed to serial port 1. Subsequently, the kernel decompressor outputs status information on this same port, as does the kernel itself. When *Neponset* is attached, however, *Angel* switches to serial port 3 and uses port 1 for debugging information.

In order to avoid requiring developers to tether their boards with two serial cables when one should be sufficient, several changes were made in both the kernel decompressor and the kernel itself. Using a method described in a later subsection, the decompressor detects the presence of *Neponset* at boot time, and configures SA-1110 UART 3 to be the target of all status and debug messages. Using the same method,[15] the kernel also detects the companion board and chooses the console serial port accordingly.

Regardless of which UART is selected for output, the port is always configured to run at 9600bps using 8-bit data, no parity, and one stop bit. Hardware

[13]The shipping configuration provides 32 megabytes in two 8M×16 SDRAMs, although sizes of 16 megabytes and 64 megabytes can be achieved using alternate components.

[14]The compressed image can be 3MB; the uncompressed filesystem image can be up to 8MB in size, including inodes and other metadata.

[15]This time, written in C rather than assembly, and running with the memory management unit enabled, thus requiring virtual mappings of the SA-1110 System Control Module registers.

flow control is not supported. The userland configuration seen most frequently on *Assabet*, assembled by Nicolas Pitre, also uses these parameters for programs such as *getty*. This allows a single terminal emulation session to successfully receive all messages output by *Assabet*, from the boot loader all the way to the *login* prompt.

3.1.4 LEDs

Assabet includes six LED indicators, two of which (D8 and D9) are general-purpose and can be manipulated directly by software.[16] During kernel bring-up, these two diodes are the only method of obtaining status information from the board, a constraint which can require some creative assembly coding on the part of the developer. Fortunately, now that the kernel runs on *Assabet*, these indicators can be assigned to some other task.

If the kernel is suitably configured,[17] the two general-purpose LEDs can indicate a regular "heartbeat" strobing pattern as well as a CPU activity monitor. The green LED D8 is assigned to the former task, while the red LED D9 is responsible for the latter. This simple status information has proven to be quite helpful for identifying system lock-ups and other troublesome events. An API — first authored by Russell King — also exists in the kernel to allow these assignments to be overridden, thus exposing the LEDs to manual control.

Perhaps more useful than the pair of general-purpose indicators found on *Assabet* is the array found on *Neponset*. Thirty-two separate LEDs are accessible through a register implemented by the CPLD on the companion board. This register appears in the address space of the SA-1110, and can be manipulated using simple word loads and stores.[18] The default value displayed on the LED array at startup time is a copy of the *Neponset* `WHOAMI` register, which contains the system ID.

[16] LED D4 indicates that the SA-1110 V_{core} is active. D10 is attached to the UCB1300[UCB] POTS modem off-hook signal. D11 indicates clipping in the UDA1341[UDA] stereo codec microphone input. D13 is used for boot diagnostics, and is also attached to the L3 bus MODE signal.

[17] `CONFIG_LEDS` must be specified, in addition to one or more of `CONFIG_LEDS_TIMER` and `CONFIG_LEDS_CPU`.

[18] To the C programmer, this is even simpler: the LEDs can be set with assignments to the symbol `LEDS`, which can be thought of as an `unsigned int`.

3.1.5 *Neponset*

Many services in the kernel require specialized configuration when the SA-1111 Microprocessor Development Module (*Neponset*) is present. Console output may switch to a different serial port, special interrupt handling services may be initialized, and device configuration for hardware on the companion board may have to occur. One solution for dealing with these special requirements might be to statically configure a kernel which would run only on systems with an attached *Neponset*, but this is clumsy. A better approach would be to detect the presence of *Neponset* at run time, and perform these additional initialization tasks as needed.

Assabet provides a mechanism for detecting ceratin aspects of the basic hardware configuration, which is documented in the *User's Guide* for that board. Through an interface named the "System Configuration Register," (`SCR`) information such as the size of SDRAM, the size of flash RAM, and the presence of optional expansion boards may be obtained. The procedure for reading this register is reviewed below.

GPIO pins 9:2 on the SA-1110 are selectively attached to 100kΩ pull-down resistors. These pins can be driven to 3V by writing `0xff` to the relevant bits in the GPIO Pin Output Set Register (`GPSR`). After precharging the pins, the pin voltages may be sensed once the levels on the pulled-down pins reach a valid logic zero state (2μs), but before the pins which are not pulled down cease to be in the valid logic one region (10μs). The values read constitute the contents of the System Configuration Register, which is, in fact, not a register at all.

The SCR is read once by the kernel decompressor, and again by the kernel during early initialization. Kernel code may test for the presence of *Neponset* at any time by using the macro *machine_has_neponset()*, which simply checks for the `SA-1111` bit in the value of the `SCR` sampled at initialization time.

Currently, there are several resources aboard *Neponset* which are not being used. These include the additional 32MB of SDRAM,[19] the PS/2 port, and the SMSC9196 ethernet controller. The initial work on *Neponset* has focused almost exclusively on the SA-1111, which is described next.

3.1.6 SA-1111

The StrongARM SA-1111 Microprocessor Companion Chip is a compact part[20] which encapsulates a number of features which might be useful on a mobile communications device. Specifically, it internalizes the majority of a Compact Flash implementation, a PCMCIA implementation, an I^2S[I2S] or AC-Link serial audio bus, PS/2 mouse and trackpad interfaces, a USB host controller, and 18 additional GPIO pins. This section covers the process of activating the SA-1111 and servicing the interrupts it generates.

It has been the experience of the author that the SA-1111 documentation was not as clear on the matter of setup as might have been hoped. In actuality, the intitialization process is straightforward, and will be described here.

Data transfers between the SA-1110 and the companion SA-1111 occur via the full 32-bit processor memory bus, using a functional block on the SA-1111 known as the System Bus Interface. Addresses presented to the System Bus Interface may be decoded[21], and directed to some other internal functional block. The datapath which allows this communication is the Register Access Bus (RAB).

In order to operate the RAB, the SA-1111 must receive a clocking signal.[22] The SA-1110 can provide such a signal, using the 3.6864MHz clock output on GPIO pin 27. To enable this clock, pin 27 is configured for its alternate function in the GPIO Alternate Function Register (`GAFR`), and set to output in the GPIO Pin Direction Register (`GPDR`). Then, a test-select bit must be set in the Test Unit Control Register (`TUCR`) to actually route the internal clock out to GPIO 27.[23] Once the clock is enabled, the `SKCR` on the SA-1111 is programmed to enable the internal phase-locked loop and system bus clocks. At this point, the remaining SA-1111 functional blocks should be accessible.

One of the most interesting functional blocks in the SA-1111 is the Interrupt Controller, which manages no fewer than 50 separate interrupt sources within the chip. These include 18 for GPIO, six for the PS/2 interfaces, three for the synchronous serial port, 11 for audio, six for USB, and three for each of the PCMCIA interfaces. All fifty of these sources share a single interrupt line back to the SA-1110. Although the SA-1111 manual recommends that this signal be routed to GPIO 0 or GPIO 1 on the SA-1110, we will see that *Neponset* incorporates a different design.

SA-1111 interrupts are enabled using a pair of Interrupt Enable registers (`INTEN0`, `INTEN1`). Once a source is unmasked using these registers, a level transition at the source generates an interrupt according to the setting of the matching Interrupt Polarity registers (`INTPOL0`, `INTPOL1`). Interrupts can be set on either a rising edge (the default), or a falling edge.[24] When an interrupt from the SA-1111 is detected, the Interrupt Status/Clear registers (`INTSTATCLR0`, `INTSTATCLR1`) can be consulted to determine the source, and then used to clear the pending interrupt from the controller.

Of course, an interrupt must be configured for the SA-1111 itself back at the SA-1110. This can be set up in the same manner as for all GPIO interrupts; polarity is set via the appropriate bit in either the GPIO Rising-Edge Detect Register (`GRER`) or the GPIO Falling-Edge Detect Register (`GFER`), and the

[19] *Neponset* SDRAM runs at $\frac{1}{4}$ the SA-1110 clock, in contrast with the clk/2 memory on *Assabet*.

[20] It uses the same μBGA packaging as the SA-1110, 15.0mm×15.0mm×1.56mm.

[21] Bits 25:9 specify a functional block within the SA-1111, with a slight twist. If the System Bus Interface registers are being accessed, bits 25:13 must be zeroed. For any other block, bits 25:13 are not decoded. Within a block, bits 8:2 specify a register.

[22] This can be confusing, because the System Bus Interface registers, such as the Control Register (`SKCR`) or ID register (`SKID`), do not require a clock. For *all other blocks* in the SA-1111, however, the clock is necessary.

[23] This part is almost outrageous: the `TUCR` is documented on the last page of the last appendix of a manual which is over 400 pages in length. The appendix begins, "The Test Unit contains a register that enables certain test modes. Some of these test modes are reserved for manufacturing test and should not be invoked by an end user."

[24] The SA-1111 manual recommends that non-GPIO interrupts use the default polarity. Interestingly, *Neponset* PCMCIA interrupts such as card detects are falling-edge.

interrupt is unmasked using the Interrupt Controller Mask Register (`ICMR`). When an interrupt occurs, the appropriate registered IRQ handler is invoked. In the case of a GPIO interrupt, the handler may consult the GPIO Edge Detect Status Regsiter (`GEDR`) to determine which pin experienced a transition. The handler then clears the interrupt, again using the `GEDR`, and services the requesting device.

There are two unfortunate design choices in the *Assabet* and *Neponset* combination that are worth noting with respect to interrupt handling. The first is that the *Neponset* interrupt signal is routed to GPIO 25 on the SA-1110, which means that the interrupt handler cannot immediately begin by servicing the board, as this pin shares an IRQ with the other pins in the set of GPIO 11–27. Some additional logic is required to ensure that a transition actually occurred on pin 25. Worse, this signal is not the SA-1111 interrupt, but rather, an interrupt shared by three devices on *Neponset*; the actual source can be decoded from the Interrupt Reason Register (`IRR`) on the companion board. The flow of execution in servicing SA-1111 interrupts is shown below.

In Figure 2, an example first-tier interrupt source attached directly to the SA-1110 via GPIO pins 0–10 is illustrated for reference. The illustration is meant to indicate that processing of the source can occur immediately, given the IRQ number, once the interrupt arrives. By comparison, handling of any SA-1111 interrupt sources requires several layers of indirection before the identity of the requesting agent is known.

Not only does this somewhat awkward interrupt architecture increase servicing latency, it also creates problems for device driver authors seeking to write code which will be portable across StrongARM board designs. Traditionally, a Linux driver requests an IRQ, and unless the IRQ is requested in "shared" mode, the driver can assume that when it is entered, the device it manages has asserted an interrupt. This does not necessarily hold for drivers running on the StrongARM, however. The IRQ requested by the device might be shared among several GPIO sources, which themselves may multiplex additional sources downstream as shown in Figure 2. Legacy drivers — especially those ported from other architectures — cannot be expected to deal directly with this organization. A solution for dealing with this problem is presented later in this report.

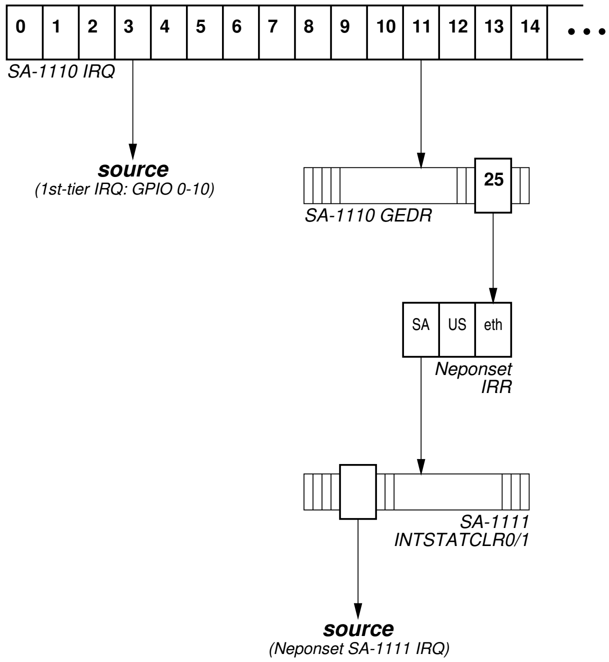

Figure 2: SA-1111 interrupt handling on *Neponset*.

3.1.7 SA-1111 Serial Audio Controller

One of the significant functional blocks in the SA-1111 which can be accessed after performing the setup procedures just described is the Serial Audio Controller (SAC). The SAC implements a number of standard control and data interfaces to external audio codecs, which are responsible for the actual analog-to-digital and digital-to-analog audio functions. Specifically, codecs which support the AC-link, I^2S, and MSB-Justified audio formats can be used, as can those which implement the I^2S and L3 bus command interfaces. The SA-1111 SAC is capable of full-duplex operation, and can either move data between the host processor and the codec using manually-loaded FIFOs, or move data directly between main memory and the codec using internal DMA engines. Audio processing may occur at sample rates up to 22.05kHz, using 16-bit, two-channel samples.

Neponset is populated by default with a Philips UDA1341 audio codec attached to the SA-1111 L3 bus interface.[25] This is the same codec found on *Assabet*, but on that board, the codec attaches directly to the SA-1110 Synchronous Serial Port (SSP, serial port 4). In the *Assabet* configuration, four GPIO pins (10–13) are consumed for clocking and framing, three more are taken for L3 functions (15, 17, 18), and yet another is lost in receiving the externally-generated sample clock (19). Adding additional complexity to the *Assabet* approach is the need for a complete software implementation of the L3 bus protocol, which manually clocks all inbound and outbound address and data words.

By comparison, the UDA1341 on *Neponset* requires fewer SA-1110 resources, and can be implemented with a simpler device driver. All SAC interrupt sources are multiplexed along the single SA-1111 interrupt signal, described in the previous subsection. The sample clock is generated internally from the SA-1111 Phase-Locked Loop (PLL) clock, which is in turn derived from the incoming 3.6864MHz oscillator on SA-1110 GPIO 27.[26] Already, this frees six SA-1110 GPIO pins and eliminates the need for an additional oscillator and clock generator.[27] The real complexity savings come from the SAC L3 bus implementation, however. To read or write command words from or to the codec, all the driver must do is load the desired target address into a control register (`L3CAR`), and in the case of a write, load a word into a data register (`L3CDR`). The driver can then poll a SAC status register (`SASR0`) to determine that the operation has completed, or enable a processor interrupt to indicate completion. In the case of a read, the returned value is placed in the L3 data register. The L3 bus can perform single or multiple-byte transfers in this manner.

The L3 interface is used to configure various features of the codec, such as input gain or bass boost. To actually transfer audio samples to or from the codec, a separate serial interface is used. These samples may be encoded using a number of formats, but in the case of the UDA1341, I^2S is the encoding chosen. I^2S uses a serial audio data line — one for each of output and input — combined with a bit clock (to drive the codec bit-sampling logic) and a `SYNC` signal to delineate left and right audio channels. Samples, which are always 16 bits per channel, are sent MSB-first, left channel followed by right channel. This encoding and the serial protocol used to transmit it are hidden completely from software by the SAC. In order to exchange samples with the codec, the SA-1110 can load or store up to eight 32-bit samples in a data register bank (`SADR`). The SAC can interrupt the processor when a configurable transfer threshold condition has been met.

A more efficient method for managing audio transfers is the one implemented in the Linux driver; on-board SAC DMA engines can move audio samples directly to or from the SDRAM bank on *Assabet*.[28] During SA-1111 initialization, just after enabling the SA-1111 clock signal, the kernel configures the memory bus request/grant mechanism needed for SA-1111 DMA. The mechanism consists of a pair of GPIO pins on the SA-1110 (21, 22) whose alternate function it is to allow the SA-1111 to claim the memory bus. Memory request support is enabled in the SA-1110 by setting a bit in the `TUCR`, discussed in the previous subsection. Because the SA-1111 will be driving the *Assabet* SDRAM directly, the shared memory controller on the SA-1111 must be configured to generate correct addresses and provide the

[25] More or less. A design error routed the L3 `MODE` signal to the SA-1111 `PWM0` pin rather than the multi-function `PWM1` pin. This error was addressed by Intel using a jumper wire to redirect the UDA1341 L3 `MODE` signal to part `J7` on *Neponset*. `J7`, of course, is electrically joined right back to `PWM0`, so this "fix" is an unfortunate no-op. Manually removing the jumper wire from `J7` and resoldering it to the nearby `J4` is necessary in order to use the L3 functions on *Neponset*.

[26] The PLL can be configured via the clock divider register, `SKCDR`; its default rate is 144MHz. The sample clock sent to the codec is generated as a function of the PLL using a divider specified in the audio clock divider register, `SKAUD`. Various dividers permit popular sample frequencies such as 8kHz, 11.025kHz, 16kHz, and 22.05kHz.

[27] Technically, *Assabet* does not use an external oscillator for the sample clock, either. Instead, a glue logic CPLD which supports the UDA1341 derives the signal from the SA-1110 memory clock.

[28] Jumpers `R51` and `R52` allow the target of SA-1111 DMA to be either bank 0 on *Assabet*, or bank 2 on *Neponset*, with the default being to select *Assabet*.

appropriate access latency. For the default *Assabet* memory load, the control register (`SMCR`) is set to use 14-bit row addresses and to expect data to be latched in response to an SDRAM read 3 memory clock cycles after deassertion of the Column Address Strobe (CAS) signal.[29] Once the shared memory interface is set up, 32-bit linear audio samples may be placed in (or retrieved from) main memory[30] as two 16-bit channels (data for the "left" channel is placed in bits 15:0, the "right" channel in 31:16), with samples having fewer than 16 bits being left-justified in the 16-bit field. Two transfer engines ("*A*" and "*B*") are available for each of the transmit (to codec) and receive (from codec) directions. A transfer begins by loading bits 25:0 of the SDRAM buffer physical address into a start register (one of `SADTSA`, `SADTSB`, `SADRSA`, or `SADRSB`), and loading the length of the transfer in bytes into the corresponding count register (one of `SADTCA`, `SADTCB`, `SADRCA`, or `SADRCB`). The engine is started by setting the appropriate bit in the control/status register for the transfer direction (`SADTCS` or `SADRCS`). When the transfer completes, the SAC can interrupt the processor, and a new transfer can be configured.

The SA-1111 documentation is unusually poor where the DMA features of the SAC are concerned. The Linux driver code owes a great deal to the experiences of developers on other operating systems, some of which strongly informed the final structure of the software. One developer observed that one or both of the "*A*" and "*B*" engines would occasionally lock up, thus requiring a watchdog timer and a facility to reset the engine. Another noticed that once a DMA engine is "enabled," it can never be "disabled." This developer also observed that the interrupt-enable bits in the DMA control/status registers are nonfunctional, as are the "transfer done" indicator bits and the "buffer in use" bits. This developer also discovered that the engines lock up if transfers are not launched in alternating sequence ("*A*," "*B*," "*A*," ...), or if FIFO depths other than the default are specified. Combined with the DMA corruption bug recently announced in a specification update, the SA-1111 SAC leaves something to be desired, at least in the current metal revision. The author hopes for a respin.

[29] These settings must always be consistent with their counterparts on the SA-1110.

[30] Unfortunately, things aren't quite this simple. A recent update to the SA-1111 specification indicates that DMA transfers to addresses beyond 1MB above the base of SDRAM can be corrupted. The workaround, to confine DMA activity to known "safe" windows in SDRAM, has not yet been added to the Linux uncached buffer request algorithm.

3.1.8 PCMCIA Card Services

In the opinion of the author, one of the more compelling features of the StrongARM SA-1100 processor is the built-in support for Personal Computer Memory Card International Association-compliant devices, also known as PCMCIA cards, or more recently, "PC cards." The range of peripherals available in this form factor is interesting, and includes ethernet adapters, wireless LAN interfaces, GPS receivers, flash RAM modules, rotating disk drives, POTS modems, speech coprocessors, and more. In addition, a follow-on standard which is a subset of PCMCIA, known as CompactFlash, provides similar functionality in a form factor smaller than a book of matches. The SA-1110, plus some external glue logic and a power controller, can support up to two slots of either type. The SA-1111 internalizes all of the glue logic necessary to support one full PCMCIA device and one CompactFlash slot.

A device driver for a given PCMCIA or CompactFlash device may be written in much the same way as a driver for a conventional device, such as a PCI ethernet controller. Such a driver would not take advantage of one of the chief advantages of PCMCIA, however: *hot-swappability*. A PCMCIA implementation has the ability to power on a card at insert time, determine its time, load an appropriate device driver, and configure the device dynamically. Correspondingly, upon card removal, any resources occupied by the device should be released, including the loadable driver.

The software infrastructure necessary to support this kind of operation is known as Card Services, and an implementation for Linux on other architectures (such as the `i386`) has been available for more than six years at the time of this writing. Card Services abstracts away the details of a particular PCMCIA implementation,[31] allowing drivers for individual cards to be written in a fairly general fashion. In the ideal case, a single Card Services device driver will work on multiple architectures without modification.

The SA-1110 contains internal logic to implement many aspects of the PCMCIA interface, but this is not to say that all SA-1110 board designs are similar in the way they choose to fully implement PCMCIA. Several aspects of the interface are left at the discretion of the designer, such as the routing of interrupts, and the control mechanism for an external power converter. The design elements which can vary among StrongARM SA-1100 implementations are presented below.

1. **Card Detect**
   When a PCMCIA or CompactFlash device is inserted into (or removed from) a slot, an interrupt is generated to notify the operating system that a card change has occurred. A StrongARM SA-1100 board designer might implement this in many ways. For example, an interrupt for each slot might be assigned to a separate GPIO pin. Perhaps the interrupts go to different GPIO pins, but these pins are in the range 11–27, so they share a single IRQ (11). Alternatively, the interrupt could go to some external logic which consolidates many signals into a single interrupt; software would have to consult a register to determine the actual source.

2. **Device IRQ**
   For I/O cards, such as an ethernet interface, the device requires the ability to notify the host that some action is necessary. This signal may be routed in any of the ways mentioned for Card Detect.

3. **Battery Voltage Detect**
   Some cards contain an on-board power source which is used to preserve the contents of memory. Such devices can notify the host when their internal battery voltage drops below the level where data integrity can be maintained. A designer may choose to route this signal as an interruptable source, as in the Card Detect and Device IRQ cases above, or may simply require software to poll the device for changes in this signal.

4. **Voltage Sense**
   When a device is inserted into a card slot, it selectively asserts a pair of signals which allow it to indicate its preferred operating voltage. Devices may make a variety of requests, such as "5V, else none" or "5V, else 3.3V, else none." There is even an "unspecified" voltage ("X.XV") available. This information may be conveyed directly to GPIO pins, to external logic such as a CPLD (where it can be read via register accesses), or to a companion processor such as the SA-1111.

5. **Write Protect**
   This signal indicates to the host that a memory device, such as a flash RAM card, will not accept new data for storage. Like the Voltage Sense pins, this signal may be routed directly to the StrongARM, or may go through external logic (such as the SA-1111).

6. **Reset**
   PCMCIA devices may be hard-reset by the socket controller, for example, as part of the normal power-up sequence. The reset signal is generally controlled by external logic (such as a CPLD), or by a companion processor (such as the SA-1111).

7. **$\mathbf{V_{cc}}$**
   The operating voltage for a PCMCIA device is indicated by the values of the Voltage Sense pins, as described above. The socket controller may apply this voltage indirectly, through external logic (such as a CPLD), or may manipulate a power converter directly.[32]

[31] In a traditional workstation design, a PCMCIA controller might reside on the system bus, and encapsulate all aspects of the implementation. In such systems, Card Services serves simply to hide which controller is being used, rather than to mask variances in how the low-level details are managed.

[32] This method is used on *Neponset*, where four GPIO signals from the SA-1111 are used to control the on-board MAX1600 power control device.

8. $\mathbf{V_{pp}}$
   The programming voltage — generally either 12V or V_{cc} — may be configured in the same manner as V_{cc} above.

Because each of these aspects of the PCMCIA implementation may be handled in at least two or three different manners, it becomes clear that the operating system needs to intervene and provide some abstraction. In the parlance of Card Services, this comes in the form of a software component called the "socket driver." The socket driver receives requests from Card Services using an API named "Socket Services," and implements the requests using the available hardware. An example request might be to configure the attribute memory space of a given slot mapped at a particular location to run at an access speed of 300ns.

With a socket driver to handle the low-level details, much of the remaining portions of Card Services can be made remarkably general. Immediately above the socket driver is the PCMCIA core, which manages resources such as memory maps and IRQs, and also manages communication with the card device drivers as well as userland configuration tools.

Card device drivers, also known as client drivers, make many of their resource requests through the PCMCIA core. This indirection frees clients from having to discover available system resources, such as memory maps, for themselves. Note that some legacy drivers which have recently been made compatible with Card Services may perform their own requests, such as interrupt registration. Such drivers still query the PCMCIA core to learn the set of available IRQ numbers.

Many configuration operations take place with the help of userland tools, and in particular, *cardmgr*. This dæmon reads a configuration file on startup, and passes a resource policy down to the PCMCIA core, which runs in kernel mode. The policy might include a list of memory ranges to map (or *not* to map), and a set of allocatable IRQs. Communication occurs via a special client driver which implements the Driver Services interface. The Driver Services client is always installed, and handles all interaction with userland.

The dæmon also maintains a database of information about specific PCMCIA devices it expects to see. Each PCMCIA card contains a Card Information Structure which describes various aspects of the device, such as manufacturer, model, and general device class. When a card is inserted, this structure is read, and the identifying information is passed up to *cardmgr*, which consults its database. A successful lookup will yield the name of a client driver, which is a kernel loadable module. The dæmon examines the filesystem, and attempts to load the module, which is then configured by the PCMCIA core running in the kernel.

Five separate components of Card Services have just been covered: the socket driver, the PCMCIA core, client drivers, the special Driver Services client driver, and the userland *cardmgr*. Initially, every one of these save for Driver Services required some degree of attention in the port to the StrongARM.

We begin with the socket driver, which had to be authored from scratch. This component was originally developed on the *Itsy/Cue* hardware which, while based on the *Brutus* SA-1100 development platform, already demonstrated significantly different design choices where the PCMCIA interface was concerned. Specifically, while both use a CPLD "register" to absorb reset signals, power control, voltage sense, and I/O status change, the register layouts are completely different. *Brutus* uses GPIO assignments for BVD, CD, and device interrupt, whereas *Itsy/Cue* leaves BVD in the CPLD (thus requiring only two pairs of GPIO pins, albeit still using different assignments from *Brutus*). Newer designs are even more different; *Neponset* uses only a single GPIO interrupt, but requires navigation of the bizarre interrupt source detection hierarchy discussed previously. Rather than consolidating control to a single CPLD register, the SA-1111 distributes PCMCIA control over several internal registers, some of which drive external, supporting hardware.

Early incarnations of the *sa1100* socket driver sought to discover a fully-general representation for the various resources which need to be managed on various boards. As it eventually became clear that no wieldy generalization was poised to present itself, an abstraction layer was added to the kernel, below which board-specific details could be hidden. This reorganization allowed the "generic" socket driver to shrink to a more manageable 1100 lines of code, with the average board-specific routines occupying about 200 lines.

In the new scheme, a StrongARM PCMCIA implementation can be supported by supplying five routines which are invoked from the "generic" driver. The first (*init*) performs any necessary GPIO setup, power control configuration, and interrupt registration which must be in place for normal operation. The second (*shutdown*) undoes the effects of *init*. The third (*socket_state*) fills in a structure for each configured socket describing the status of various signals such as card detection, battery voltage detection, voltage sense, write protection, and so forth. The fourth (*get_irq_info*) returns, for the requested socket, the IRQ assignment which should be requested for service to a device in that socket. The last (*configure_socket*) allows Card Services to configure V_{cc} and V_{pp} for a given socket, as well as assert reset. At the time of this writing, the kernel includes versions of these routines for four distinct PCMCIA implementations.

With the abstraction layer in place, the "generic" socket driver need only implement the Socket Services API, occasionally receiving requests from Card Services (and passing some of them to the abstraction layer), and occasionally receiving interrupts from the abstraction layer (and forwarding them up to Card Services). Examples of requests issued by Card Services include *set_socket*, which configures the per-socket V_{cc}, V_{pp}, reset and so forth; and *set_io_map*, which configures PCMCIA I/O memory mapping and access speed.[33] Examples of interrupts received by the socket driver include those for card detection or a change in battery charge state. When any interruptable event comes in, a bit field is filled in (subject to an event mask) to describe the change in status. This field is passed up to Card Services via a callback that was arranged at initialization time. Note that the one interrupt *not* serviced by the socket driver is that for the card itself; this interrupt is requested by the card device driver directly.

Continuing with the remaining Card Services components which required modification, the PCMCIA core ended up requiring very few alterations. Card Services, like the Linux kernel, contains a number of legacy assumptions about the nature of I/O access. These include constraints on the kind of addresses used to reach I/O (in particular, they assume 16-bit ISA "ports"), and the facilities available for performing such access (such as the $x86$ I/O instructions `inb`, `outb`, `inw`, and `outw`). This can present an awkward fit for a machine which maps I/O directly into a 32-bit address space, and which can access this memory via normal loads and stores, such as the StrongARM. The address width problem was more or less straightforward to correct, while the I/O "instruction" problem required slightly more effort. The Card Services code invokes wrappers for the $x86$ I/O instructions (*e.g.*, the C routine *inb()* for `inb`), as is common for kernel drivers. The ARM architecture port of the kernel implemented versions of these wrappers that were meant to simplify the task of hacking together a PCMCIA client driver to work directly on the StrongARM (this was necessary before the port of the full Card Services suite). The ARM wrappers assumed that the routines would always be used to access I/O memory in PCMCIA slot 0, and so the wrappers performed arithmetic on the target addresses they were passed. When the PCMCIA core was corrected to properly handle 32-bit addresses, the ARM wrappers were also revised to no longer make any assumptions about the locations they were to manipulate. The effects of this latter change were somewhat surprising.[34]

Several changes were made to the PCMCIA core early in the porting process which involved *cardmgr*, the userland configuration dæmon. These were later backed out of the patch set, but will be discussed here

[33] The locations of the PCMCIA memory spaces are identical across all SA-1100 processors, and the access speed configuration interface is nearly identical between the SA-1100 and SA-1110. As such, this particular request can be serviced entirely by the "generic" driver, without the aid of the abstraction layer.

[34] For example, a spurious $x86$-specific operation in the initial RAM disk code which used I/O instructions to disable a floppy disk controller was being executed during boot. This was never noticed on the ARM because *outb()* applied an offset to the target address which directed the memory write into the middle of PCMCIA 0 I/O space (which was disabled, anyway). When *outb()* was fixed, this became a write to an unmapped memory location near address `0x0`... oops.

for historical account. One rôle of *cardmgr* in Card Services is to read a user configuration file which describes certain resources available for use by PCMCIA software; this information is then delivered to the PCMCIA core running in the kernel. The resources managed by the dæmon include IRQ assignments, I/O ranges, and memory ranges. The original port of Card Services to the StrongARM added the ability to separately manage common memory and attribute memory ranges. Also added were provisions for a number of register addresses, bit masks, and other items necessary for accommodating the range of variation in StrongARM PCMCIA implementations. The goal of this approach was to produce a fully-generic set of kernel modules which would work on any StrongARM SA-1100 board without recompilation; all configuration would happen via the userland configuration tool. In the end, this approach proved to be less wieldy than the per-board abstraction layer described earlier. The various memory ranges do not change across StrongARM SA-1100 processors, so userland management of them is not necessary. The other resources, such as IRQ assignments, register locations, and so forth, are so numerous (more than thirty such features exist) that a substantial redesign of the Card Services resource system — needed only for the ARM architecture — would be required. The final design, therefore, goes nearly to the opposite extreme by ignoring the userland configuration entirely. The goal of kernel portability across multiple boards is now being approached in another way: compile-time macros and run-time tests enable various per-board features.

Concluding the discussion of Card Services components are Driver Services and the client drivers. Driver Services is a special case of a client driver which has the distinguishing ability to communicate with the userland *cardmgr*. No direct changes to the Driver Services module were necessary to support the StrongARM. In general, client drivers have tended to require only a single change: expansion of the internal data structure used to store I/O addresses from a 16-bit data type to the 32-bit field needed to represent StrongARM virtual addresses. Devices which have been confirmed to work using Card Services on StrongARM-based systems include the Lucent WaveLAN/IEEE 802.11 PCMCIA device (first using a port of the Lucent proprietary driver, then using the open source *wvlan_cs* driver), the IBM Microdrive (using *ide_cs*), the Socket Communications LP-E CF+[LPE] ethernet device (using a modified version of *pcnet_cs*; see below), and other CompactFlash flash RAM devices (using *ide_cs*). In most cases, once a driver is made to function on one StrongARM SA-1100 board, it should be able to support its device on other boards without modification.

3.1.9 *Assabet* CompactFlash

Assabet provides one type-II CompactFlash slot, suitable for accepting the included Socket Communications LP-E CF+ ethernet card, or a larger device such as the IBM 1GB Microdrive. The CompactFlash interface is mapped to PCMCIA slot 1 on the SA-1110, and can only be used when *Neponset* is not attached.[35] The control interface for CompactFlash on *Assabet* consists of four GPIO pins and three pins in the CPLD Board Control Register (`BCR`). The control interface for the slots on *Neponset* is completely different, a fact which causes the User's Guide for *Assabet* to mention, "Two versions of the CF drivers are required for the SA-1110 Development Platform, one for the SA-1111 Development Module and one for the SA-1110 Development Board." As we will see, Card Services makes this statement untrue.

CompactFlash initialization on *Assabet* begins by asserting the `CF_Bus_On` bit in the `BCR`. Next, the four GPIO lines are configured. Briefly, these are the card interrupt signal (`CF_IRQ`, GPIO 21), the Card Detect signal (`CF_CardDetect`, GPIO 22), and the Battery Voltage Detect signals (`CF_BVD1` and `CF_BVD2`, GPIO 25 and GPIO 24, respectively). All four signals will be used as inputs, a characteristic which is configured in the GPIO Pin Direction Register (`GPDR`).

The SA-1110 can generate software interrupts based on level transitions at the GPIO pins. IRQ handlers may be invoked at a rising edge, falling edge,

[35] The bus transceivers on *Assabet* are dual-purpose: when *Neponset* is present, the transcievers are redirected to drive the daughterboard interface instead of the CompactFlash slot. Damage can result if the transcievers are made to drive both interfaces at once.

or both based on the values of the GPIO Rising-Edge Detect Register (`GRER`) and GPIO Falling-Edge Detect Register (`GFER`). This flexibility was not anticipated in the Linux IRQ request facility, so some extra work is required to properly set up an interrupt handler for devices such as the CompactFlash interface. After trying several approaches, the current solution — refined outside of Carnegie Mellon — maintains a set of GPIO pins which should trigger rising edge interrupts, and another set for falling edge interrupts. These sets are configured prior to issuing the normal *request_irq()* operation, and are loaded into the GPIO edge detect registers during normal interrupt processing. In the case of *Assabet*, all four GPIO lines have their edge detection policy set during initialization, but only three (`CF_CardDetect`, `CF_BVD1`, `CF_BVD2`) are actually registered. The CompactFlash device IRQ will be registered at card initialization time by the card driver itself using the normal Linux mechanism.

After initialization, the generic *sa1100* socket driver makes a number of *socket_state* or *configure_socket* requests, possibly in response to a CompactFlash-related interrupt or some other Card Services event. Returning the state of the CompactFlash slot is straightforward; the GPIO Pin-Level Register (`GPLR`) contains the current states of the Card Detect and Battery Voltage Detect signals, which can be copied directly into the state descriptor. Because *Assabet* does not deliver Write Protect status to the processor, and the power supply can only provide 3.3V to the card, no additional information can be returned to the socket driver beyond the various Detect signals. To configure the socket, the kernel first disables interrupts, then samples the last value written to the Board Control Register, which is write-only. Based on the configuration structure passed, the `CF_PWR` (3.3V supply) and `CF_RST` (reset) bits are adjusted, and the resulting value is written to the `BCR`, after which interrupts are re-enabled. The V_{pp}, output enable, and speaker enable features are not available for configuration on *Assabet*.

Because *Assabet* ships with a CompactFlash device for which Linux support has been added, it seems appropriate to discuss the driver for that device here. The Socket Communications LP-E CF+ ethernet card is based around an NE2000-clone controller, for which support previously existed in the kernel. The *8390* driver provides the core support for this controller, including most of the hooks into the Linux ethernet device code. A wrapper for this driver, *pcnet_cs*, implements the necessary PCMCIA Card Services operations and performs the various dynamic configuration tasks associated with PC card versions of this controller. In order to properly configure the device, a driver must first read the ethernet Medium Access Control (MAC) address from the CompactFlash card attribute memory, and then program the controller using this address. The location of the MAC address is variable across card implementations, and Socket Communications was unwilling to provide documentation for their device. A brute force search method was sufficient to discover the address location, and an entry was added to the hardware address search table in the driver to support the LP-E CF+ card.

A more serious problem associated with the Socket Communications card was discovered by an engineer at Intel, for which a software workaround has been added to the Linux driver. The symptom consists of corruption on the last octet of an odd-length transfer from the card to the processor during a frame receive, when the card is programmed to transfer 16-bit quantities. The corruption is fatal to TCP-dependent applications, or indeed, any protocol with sufficiently sensitive error detection. The current fix is to detect these odd-length buffers, and drop the card into 8-bit transfer mode prior to moving the final octet.

3.1.10 *Neponset* PCMCIA/CompactFlash

As mentioned previously, the PCMCIA and CompactFlash slots on *Neponset* are accessed in a completely different fashion from the *Assabet* slot. The GPIO pins assigned to CompactFlash on *Assabet* are instead responsible for the SA-1111 shared memory interface and daughterboard interrupts. Card configuration occurs largely via the PCMCIA functional block on the SA-1111 instead of the *Assabet* `BCR`, with some power control features moved to the *Neponset* Control Register (`NCR_0`). Despite these significant differences, it is not necessary to implement separate client drivers for a CompactFlash device running on *Assabet* and *Neponset*. The socket driver abstraction facility masks these implementation issues completely.

PCMCIA and CompactFlash initialization begins by assuming that access to *Neponset* has been set up, and that the SA-1111 Register Access Bus has been properly activated. GPIO block A on the SA-1111 includes four signals which are connected to the MAX1600 power switching network on *Neponset*. As these willl be used exclusively as outputs, their direction is configured using the block A Data Direction Register (`PA_DDR`). The MAX1600 is programmed to standby mode by clearing all four V_{cc} signals (by writing the block A Data Write Register, `PA_DWR`) as well as the two V_{pp} signals (by writing to `NCR_0`).

Like the SA-1110, the SA-1111 can assert a software interrupt based on level transitions[36] for a number of signals. Of the nearly 50 interruptable signals available, six will be of interest to the PCMCIA implementation: `S0Readynint`, `S0CDValid`, `S0_Bvd1Stschg`, and their slot 1 equivalents. The SA-1111 manual notes that non-GPIO interrupts — such as the PCMCIA signals — should assert on a rising edge. Interestingly, on *Neponset*, all six signals are properly asserted on their falling edge, instead. Assertion polarity is set by directly accessing the SA-1111 Interrupt Controller registers `INTPOL0` and `INTPOL1`.[37] With the polarity configured, interrupts for the Card Detect and Battery Voltage Detect signals are registered using *request_irq()*.

The *Neponset* versions of *socket_state* and *configure_socket* have the same responsibilities as their *Assabet* equivalents, but enjoy control over more features. All state information for either slot can be sampled through the SA-1111 PCMCIA Status Register (`PCSR`). For each slot, the following information is returned: Card Detect status, Card Ready status, both Battery Voltage Detect signals, Write Protect status, and the 3.3V and "X.XV" Voltage Sense signals. Socket configuration is slightly more complex, and begins by sampling three registers: the PCMCIA Control Register (`PCCR`), `NCR_0`, and `PA_DWR`. V_{cc} can be configured to either 0V, 3.3V, or 5V for either slot by first programming the `PCCR` to float the socket control lines (in the case of 0V), or by setting the appropriate one of the `S0PSE` or `S1PSE` bits (for 3.3V or 5V). Power is actually switched to the sockets by asserting the appropriate signals attached to the block A GPIO lines, controlled by `PA_DWR`. In a similar manner, V_{pp} to either socket is configured on the MAX1600 by two signals accessible through the *Neponset* CPLD. Bits `A0VPP` and `A1VPP` in `NCR_0` switch the 12V supply to slot 0 (the PCMCIA slot), but switch 0V to slot 1 (CompactFlash devices do not accept a 12V programming voltage). The last configurable PCMCIA setting is the socket reset signal, which is accessible through the `S0_RST` and `S1_RST` bits in the `PCCR`. Once all of the new power and reset selections have been processed, new values are written to each of the `PCCR`, `NCR_0`, and `PA_DWR`.

Unhappily, there is an exception to the previously offered claim that client drivers require no modification when moving to *Neponset* from *Assabet*, or from any other StrongARM SA-1100 board. It has been discovered that the Socket Communications LP-E CF+ card — and possibly others — cannot transfer neither frame headers nor frame data to the processor in 16-bit word mode. A runtime hack has been added to the *pcnet_cs* driver to disable the use of word mode, requiring transfers to occur one octet at a time. This has serious implications for the effective bandwidth to and from the card. Unfortunately, an insufficient number of cards have been tested on *Neponset* to determine at this time whether or not this is a fault of the LP-E device.

3.2 External Work

The Linux kernel in general is a massive distributed software development effort, aided by thousands of developers worldwide. Hundreds of core features which are critical to the usefulness of Linux on *Assabet* were put in place by these contributors over the last decade, such as the process scheduler, file sys-

[36] Note that the only options for polarity are rising edge or falling edge; it is not possible to configure assertion for both kinds of transition simultaneously.

[37] There are too many interrupt sources here to fit into a single register. The interrupt enable, polarity, and status/clear registers all appear in pairs, with sources 0 through 26 appearing in the first register, and sources 32 through 54 in the second.

tems, and serial terminal drivers. To properly acknowledge all of this work would require volumes of text, so this section aims to identify portions of the kernel which relate specifically to *Assabet* that have been contributed recently by developers not affiliated with Carnegie Mellon.

3.2.1 Interrupt Architecture

As discussed earlier in this report, interrupt organization on the StrongARM SA-1100 is perhaps more complex than might be considered ideal. On the SA-1110, some general-purpose I/O pins receive their own IRQ, while others share a common IRQ and require software to consult a register to identify the source. When the SA-1111 is added, more levels of indirection are added in order to consult the interrupt source registers on that chip. If the SA-1111 is part of a board like *Neponset*, there may be yet additional indirection required just to identify the companion chip as one among many potential requestors.

Historically, this has not been a problem, as many drivers written for Linux running on the StrongARM could be aware of this architecture, and behave accordingly. With the arrival of PCMCIA Card Services, however, the prospect arose of dozens of client drivers appearing which were written in a general fashion, with no expectation of such an unusual organization. A typical client driver issues a query to Card Services for the IRQ number it should request of the kernel, and after a successful IRQ request, the client assumes that the only work performed by its interrupt handler is that which relates specifically to the card itself. No provision exists for interrupt source lookup, or for any additional "cleanup" of the hardware after interrupt processing completes.

To avoid requiring these otherwise portable drivers to support every possible StrongARM board, the IRQ management code in the kernel was improved by Nicolas Pitre to simplify interrupt usage by drivers. The SA-1110 supports 32 separate IRQs, twelve of these associated with the GPIO interface. IRQs 0–10 map directly to GPIO pins 0–10, with the remaining 17 GPIO sources shared among IRQ 11. Using the new IRQ system, software can request IRQ numbers 32–48, which are not "real" processor interrupts, but instead cause a proxy handler to be registered for IRQ 11. At runtime, when a shared interrupt comes in on IRQ 11, the `GEDR` is consulted, reset, and the registered interrupt handler is invoked just as though it were a first-class interrupt.

IRQ support for the SA-1111 on *Neponset*, initially added to this framework at Carnegie Mellon, takes the same approach. New IRQ numbers 49–104 have been added for the interrupt sources on the SA-1111. When one of these interrupts are requested, several proxy handlers are installed. The first, for the shared IRQ 11 GPIO lines, is required because the *Neponset* interrupt is on GPIO 25. The second, for IRQ "46," is a proxy for the three interrupt sources on *Neponset*: an ethernet controller, a keyboard encoder, and the SA-1111. This proxy reads the *Neponset* `IRR` and dispatches a handler for the requesting device. The last proxy, for SA-1111 interrupts, reads and resets the `INTSTATCLR0` and `INTSTATCLR1` registers on the SA-1111, then dispatches handlers for the active interrupt sources.

The end result of this organization is that almost any interrupt source on *Assabet* or *Neponset* can be requested by software using the standard Linux *request_irq()* service, albeit with perhaps a higher IRQ number than is typical. As mentioned previously, because the SA-1110 and SA-1111 allow flexibility in configuring the kinds of level transitions that constitute an interrupt condition, this approach cannot mask the underlying implementation completely. Software must still specify using a separate interface — either through routines associated with IRQ handling, or by accessing the polarity registers directly — which interpretation should be used. In practice, this has not posed a portability problem, as the correct convention is generally known at startup time, and can be configured in board-specific initialization code.

3.2.2 Memory

This section is provided more to mention an aspect of the code which will be interesting to pursue in the future, but which has not been tested as of this writing. Nicolas Pitre has moved *Neponset* detection to an early part of the kernel setup process in order to potentially configure the 32MB of SDRAM provided on that board. At present, this memory is not available to the kernel, but enabling this bank would be interesting for two reasons. First, memory on *Neponset* is clocked at half the speed of the *Assabet* SDRAM, which would result in non-uniform access times. Second, some devices are quite specific about which RAM bank they will access. For example, the SA-1111 performs DMA transfers to only one of the two boards, the selection of which is based on hardware jumpers. The Linux non-cacheable buffer allocation implementation does not provide the ability to express constraints on acceptable memory regions.[38] Adding support for this functionality is a to-do item at the time of this writing.

3.2.3 Audio Codec

The SA-1111 Serial Audio Controller support described previously could not have been attempted without the availability of a UDA1341 driver, authored by Nicolas Pitre. The original driver was written for the part as it is connected on *Assabet*, using the GPIO-based L3 bus implementation. When running against the SA-1111, the driver does not make use of the L3-related code, but is able to leverage the existing L3 read/write API. Similarly, the programming interface for SA-1111 DMA is mimiced from the SA-1110 version so as to preserve the code flow in the rest of the driver. Much of the remaining driver code, such as the implementation of the Open Sound System[OSS] API, and the various UDA1341 command register programming features, remain unchanged from the version which works on *Assabet*.

3.2.4 DMA

Data may be transferred between system memory and any of the five SA-1110 serial ports using any of six internal DMA engines. Like the engines in the SA-1111 Serial Audio Controller, an "*A*" and "*B*" transfer can be configured for each engine. Transfers can be issued either to or from the serial controller in 8- or 16-bit quantities, and the engines can perform endianness conversion in flight. The DMA controller is capable of interrupting the processor upon completion of any transfer.

Nicolas Pitre has added an API to the kernel which supports the DMA controller. Calling code passes a register block to *sa1100_start_dma()*, through which the control registers for one of the six engines are accessible. The service routine selects one of the "*A*" or "*B*" transfers to initiate, and then loads the appropriate registers with the appropriate flags, target address, and transfer count. It is the responsibility of the calling code to handle the interrupt associated with the engine being used. Generally, such code will dequeue a fresh buffer for transfer, and re-invoke the DMA start routine. As mentioned previously, the SA-1111 SAC DMA interface is sufficiently similar that a counterpart API for that chip was easily added alongside the SA-1110 code.

3.2.5 LCD Framebuffer

The SA-1110 includes a special DMA engine which resides directly on the internal ARM data bus, which exclusively serves to transfer graphics data from an SDRAM framebuffer to an external liquid crystal display. The LCD controller is quite flexible, permitting displays of up to 1024×768 pixels at a 16-bit color depth. *Assabet* ships with a Sharp 3.9-inch LCD touchscreen, which supports either 8-bit or 16-bit color at 320×240 pixels. When running at the lower color depth, a CPLD on *Assabet* maps the 8-bit pixel data to the RGB444 (12-bit) format expected by the display. At the 16-bit depth, the RGB565 data output by the StrongARM is mapped to the 18-bit (RGB666) encoding used by the LCD. Which translation the CPLD performs is governed by the setting of the `LCD12or16` bit in the Board Control Register, with LCD power controlled by the `LCD On` bit.

Support for the CPLD interface, as well as palette configuration and other *Assabet*-specific details have all been added to the generic SA-1100 framebuffer driver. The result enables the use of Nano-X, XFree86, and other graphical display servers, as well as the standard Linux text console. Principal contrib-

[38] Unfortunately, such constraints are necessary even without considering the *Neponset* SDRAM. As mentioned earlier, a recent Intel errata documents DMA corruption to certain regions of memory regardless of their bank assignment.

utors to this effort include Jeff Sutherland, Tak-Shing Chan, and Nicolas Pitre.

3.2.6 Touchscreen

The Sharp 3.9-inch LCD module includes a touchscreen with a standard 4-wire resistive output. On *Assabet*, this output is directed to a Philips UCB1300 ADC. The kernel presently includes several touchscreen drivers, but an effort is under way to standardize the form of these drivers within the kernel.[Tou] The current version suitable for use on *Assabet* was authored by Tak-Shing Chan, based largely on a previous driver by Peter Danielsson.

A ARM Linux Patches

One of the more confusing aspects of Linux development on *Assabet* is the bewildering array of patches to the source code needed to produce a working kernel. At one point, as many as four patches to the original source have been required to achieve an up-to-date image of the software. This section attempts to explain the patch hierarchy and help developers locate the various components needed to perform useful work. The tour begins with the basic kernel distribution, and proceeds in the direction of increasing *Assabet* support. As will be discussed later, developers should actually collect the components in the opposite order.

1. **Kernel Distribution**

   The official source for the Linux kernel is available via anonymous FTP from `ftp.kernel.org`, and can be had in *gzip* and *bzip2* formats. In general, the Linux kernel uses a version numbering convention which differentiates "production" releases from "development" releases. This convention is based on the minor release number, such as the "4" in "2.4." When the minor release number is even, the release is intended for production use. When this number is odd, the release is alpha-quality at best, and should be used with caution. Having said this, it is generally true that the most recent StrongARM support is only added to the newest kernel, regardless of the release version. The earliest kernel for which *Assabet* support was added was 2.3.99, and all development is currently being done on version 2.4.0.

   As a side note, the primary maintainer for the official distribution is Linus Torvalds. Recently, while Linus was away on holiday, ownership was passed to Alan Cox, who maintains a set of patches against the main kernel tree. These patches are identified with the initials of Mr. Cox, so an example patch against kernel *2.4.0-test1* might be *2.4.0-test1-ac6*. This patch series is also available from `ftp.kernel.org`, under `/pub/linux/kernel/people/alan/`. In general, however, these patches are not used for ARM Linux development.

2. **ARM Linux Patch**

   Support for a number of machines based around ARM processors is maintained in the Linux Kernel by Russell King. Available via anonymous FTP from `ftp.arm.linux.org.uk` under `/pub/linux/arm/source/kernel-patches/` are patches which usually apply directly to the kernel distribution mentioned above. The naming convention for these patches extends the base kernel version with an identifier for the author, as in the example of the Alan Cox series above. For instance, a suitably patched version of kernel *2.4.0-test5* might be *2.4.0-test5-rmk1*.

3. **StrongARM SA-1100 Patch**

   The StrongARM family of processors is considered to be a separate architecture within ARM Linux. Support for StrongARM-based machines was originated by Russell King, for the SA-110 processor, but has been extended by Nicolas Pitre to include the SA-1100 and derivative processors. Patches for the SA-1100 family extend the ARM patches just described, and are available available from `ftp.arm.linux.org.uk`, this time under `/pub/linux/arm/people/nico/`, these patches follow the naming convention established previously. Extending the last example, such a kernel might have the final version *2.4.0-test5-rmk1-np1*. All viable StrongARM SA-1100-related kernel changes made by ARM Linux developers are folded into this series of patches.

4. **Wearable Group Patches**

   With the stabilization of *Assabet* and PCMCIA support in the current StrongARM SA-1100 patch series, the frequency with which the Wearable Group at Carnegie Mellon issues patches has decreased. Occasionally, "bleeding edge" patches are released which may not have reached the other patch levels. For example, support for *Assabet*, *Neponset*, the SA-1111, PCMCIA Card Services, and several PCMCIA client drivers have all been released by the author prior to their inclusion in the StrongARM SA-1100 patches.

   Information about, and patches implementing, support for *Assabet* and PCMCIA on the StrongARM is available on the World Wide Web at `http://www.cs.cmu.edu/~wearable/software/`.

When assembling a kernel tree for use on *Assabet*, it is generally appropriate to begin from the StrongARM SA-1100 patch (maintained by Nicolas Pitre), and work backwards towards the kernel. The ARM Linux and StrongARM SA-1100 patches are necessarily released after the corresponding kernel distribution, so it is usually possible to find versions of the base kernel which are newer than any available ARM patch. Unless a particular StrongARM SA-1100 patch includes significant feature additions, it is usually true that newer patches are more robust than older ones.

B Patch History

This section documents the patches released by the Wearable Group at Carnegie Mellon concerning *Assabet* and PCMCIA Card Services. Nearly all of the kernel patches are now part of the master StrongARM patch. All of the following patches were developed by the author of this report, and are distributed under the terms of the General Public License.

1. *linux-2.3.99-pre3-rmk1-np3-assabet1* (13/04/00)

   This was the first patch to include support for *Assabet*. Initial serial console support and memory configuration, based heavily on the code for *Brutus*, were provided. A bug caused the console to be come inaccessible following userland startup, although in retrospect, this was most likely due to a simple data rate mismatch in the userland *getty* terminal configuration. Support for the two general-purpose LEDs was present. The bulk of this 142-kilobyte patch was occupied with the task of converting the kernel nomenclature from "SA-1100" to "SA-11x0", a change which was ultimately rejected from the main StrongARM SA-1100 patch.

2. *pcmcia-cs-3.1.14-arm-4* (27/04/00)

   The fourth in a number of patches against the standalone PCMCIA Card Services for Linux distribution, this release began support for *Assabet* by moving from kernel 2.0.30 — found on the *Itsy/Cue* — to 2.3.99. In the process, the socket driver included in this release began to use an early version of the in-kernel PCMCIA abstraction facility, thereby reducing configuration complexity. (This patch was half the size of the one preceding it.)

3. *pcmcia-cs-3.1.14-arm-5* (03/05/00)

   This release continued to wrestle with the Socket Communications LP-E CF+ ethernet card, unsuccessfully. A *procfs* node was added to display the state of the socket controller, which is useful for debugging purposes.

4. *pcmcia-cs-3.1.14-arm-6* (05/05/00)

   Support for the LP-E card grows nearer, but DMA transfers during receive interrupt processing still fail.

5. *linux-2.3.99-pre8-rmk1-np1-jd1* (15/05/00)

   PCMCIA Card Services for Linux is somewhat unusual in that it involves a number of loadable kernel modules, yet it is not distributed as part of the kernel source tree. As a result, the in-kernel aspects of Card Services cannot be easily debugged on a machine such as *Assabet*, because a complete RAM disk containing new versions of the modules must usually be transferred to the board every time a code change is performed. Meanwhile, Linus had chosen to fold the Card Services modules into the 2.3 kernel series, thus enabling them to be compiled statically into the kernel. This was attractive to ARM Linux developers, who wanted to see the new *sa1100* socket driver available in the kernel tree itself.

The Card Services code which appears in kernel 2.3.99 gives the appearance of having been hijacked from its original developer, David Hinds. Some of the APIs have been changed, and the code itself is somewhat stale, appearing to have come from Card Services 3.1.10. As a result, many of the fixes originally needed to support the ARM in Card Services had to be duplicated in the kernel, such as support for fixed memory maps, large IRQ numbers, and so forth.

The Socket LP-E CF+ card finally works.

6. *linux-2.3.99-pre8-rmk1-np2-jd1* (17/05/00)

   Bug fixes for *pcnet_cs*, the low-level PCMCIA interface, and interrupt handling for card insertions. Included Erik Bunce's patch for CompactFlash ATA/IDE devices, such as the IBM Microdrive.

7. *linux-2.3.99-pre8-rmk1-np5-jd1* (30/05/00)

   Added boot-time detection of *Neponset* in both the kernel and the kernel decompressor. Fixed serial console support to use UART 3 when *Neponset* is present. Includes support for *Neponset* LEDs, and can activate the SA-1111 Register Access Bus. Adds support for SA-1111 interrupt handling and the SA-1111 PCMCIA Controller, including the MAX1600 power switching network.

8. *linux-2.3.99-pre8-rmk1-np5-jd1a* (30/05/00)

   Companion patch to the previous item, required for the *pcnet_cs* driver to work on *Neponset*. Causes the Socket LP-E CF+ card to transfer buffers in 8-bit mode, not 16 bits at a time. Corruption is observed when this patch is not applied.

9. *linux-2.4.0-test1-ac7-rmk1-np6-jd1* (16/06/00)

   Small correction to the interrupt handling facility in the StrongARM SA-1100 patch, which attempted to address a system lockup which had been observed. The patch, while able to illustrate the actual problem, still contains a logical error which could potentially still permit problems in other circumstances.

10. *linux-2.4.0-test2-ac1-rmk2-np3-jd1* (14/07/00)

    Small correction to the revised SA-1100 serial driver which fixes a parity configuration problem. Also corrects a kernel configuration omission which prevented SA-1111-related code from being enabled.

11. *linux-2.4.0-test5-rmk1-np1-jd1* (02/08/00)

    Introduces support for the SA-1111 Serial Audio Controller, and extends Nicolas Pitre's UDA1341 driver to work on *Neponset*. Occasional DMA lockups and audio corruption are observed, which may be explained by a recent Intel errata listing.

12. *linux-2.4.0-test6-rmk5-np2-jd1* (29/08/00)

    Begins conversion of PCMCIA socket driver memory management to on-demand mapping requests, and eliminates the use of static memory allocations for PCMCIA I/O, attribute, and common memory spaces.

Acknowledgements

Special thanks go to Nicolas Pitre, architecture maintainer for the StrongARM SA-1100 port of ARM Linux. Thanks also go to Jeff Sutherland, Andrew Pfiffer, Erik Bunce, Wayne Peters, Mark Murawski, and Ryan Zoschg for their very helpful advice and feedback during the development process.

References

[ARMa] Advanced RISC Machines. http://www.arm.com/.

[ARMb] The ARM Linux Project. http://www.arm.linux.org.uk/.

[Ass] Intel StrongARM SA-1110 Microprocessor Development Board User's Guide. http://developer.intel.com/design/strong/.

[Com] CompactFlash Association. http://www.compactflash.org/.

[I2S] I^2S Bus Specification. http://www.semiconductors.com/.

[Itsa] The Itsy Pocket Computer. http://research.compaq.com/wrl/projects/itsy/.

[Itsb] The Itsy/Cue Computer. http://www.cs.cmu.edu/~wearable/itsy/.

[Lin] Linux. http://www.linux.org/.

[LPE] Socket Communications Low-Power Ethernet CF+ Card. http://www.socketcom.com/lpecfmp.htm.

[Mic] IBM Microdrive. http://www.storage.ibm.com/.

[Nep] Intel StrongARM SA-1111 Microprocessor Development Module User's Guide. http://developer.intel.com/design/strong/.

[OSS] The Open Sound System. http://www.opensound.com/.

[PCM] Personal Computer Memory Card International Association. http://www.pc-card.com/.

[SA1a] Intel StrongARM SA-1110 Microprocessor Advanced Developer's Manual. http://developer.intel.com/design/strong/.

[SA1b] Intel StrongARM SA-1111 Microprocessor Companion Chip Developer's Manual. http://developer.intel.com/design/strong/.

[Tou] Specification for Generic Touch Screen Driver. http://handhelds.org/projects/ts_spec.html.

[UCB] Philips Semiconductor UCB1300 Advanced Modem/Audio Analog Front-End. http://www.semiconductors.com/pip/UCB1300BE.

[UDA] Philips Semiconductor UDA1341TS Audio Codec. http://www.semiconductors.com/pip/UDA1341TS.

[Wav] Lucent WaveLAN (ORINOCO). http://www.wavelan.com/.

[Wea] The Wearable Group at Carnegie Mellon University. http://www.cs.cmu.edu/~wearable/.