Unix for Nomads: Making Unix Support Mobile Computing

Micheal Bender, Alexander Davidson, Clark Dong, Steven Drach, Anthony Glenning, Karl Jacob, Jack Jia, James Kempf, Nachiappan Periakaruppan, Gale Snow, Becky Wong[1]
Nomadic Systems Group,
Sun Microsystems Computer Corp.
2550 Garcia Ave., Mail Stop MTV17-08
Mountain View, CA, 94043

Abstract

Traditionally, the Unix operating system[2] has been associated with deskbound machines tethered to the wall by a power cord and an Ethernet cable. Making Unix support a more nomadic model of computing requires changes in the entire system, from the kernel level through the user command set of applications. In this paper, we present the results of an experimental prototype development effort targeted at supporting a nomadic computing model in Sun's Solaris 2 SVR4-based platform[3]. The development involved enhancements in four primary areas: kernel changes to support power management and checkpointing of system state, drivers and other kernel support for the new PCMCIA bus standard, support for serial line networking, and a new electronic mail application designed specifically for accessing mail over slow serial connections. The paper discusses enhancements and modifications to the design of standard Solaris system components in each of these areas.

1. Introduction

An implicit assumption in the design of many Unix platform software components is that the workstation or server will remain tethered to a particular physical location by its need for power and an Ethernet connection. Newer, low power chip and system designs have enabled the manufacture of small form-factor, battery-powered portable computers. These computers are much more mobile than the machines on which Unix traditionally runs. Such machines may be used both in an office setting and while on the road, so that they become the primary machine for some users. A *nomadic computing model*, where users set up their computer and work somewhere then move the machine to a different physical location and continue working, is fundamentally different from the traditional deskbound model which Unix currently supports. Design modifications to a Unix platform for supporting nomadic computing require a system approach, since the nomadic usage model affects everything from the kernel through user applications.

The main factor driving kernel changes for nomadic use is power. Mobile computers tend to have tighter constraints on power consumption than desktop machines. These constraints derive primarily from the limited performance of existing batteries. Since battery technology is improving slowly relative to processor and memory technology, hardware and software for saving power in nomadic computers is essential to increasing the available compute time with limited battery life. In addition, nomadic usage patterns involve frequent power cycling, so that long start up and shut down times are less tolerable than in desktop machines. Mobile users want the ability to simply shut the machine off and later start it up again, having it restored to exactly the same state as they

1. Author's names are in alphabetical order.

2. Unix is a registered trademark of Unix System Laboratories.

3. In this paper, SunOS 5.x refers to SunSoft's implementation of Unix System V Release 4 (SVR4). Solaris 2.x refers to SunSoft's SunOS 5.x-based application development and delivery environment, which includes the OpenWindows window environment and the Tooltalk application integration environment.

left off so they can begin work immediately, because their work is frequently interrupted by movement. While similar functionality has been added to MS-DOS[4] and SunOS 4.1 [Tadpole93], to our knowledge, our work is the first attempt to add power management and checkpoint/resume to a version of SVR4.

Nomadic users may have limited access to the standard Ethernet connections which many deskbound machines require. A wide variety of connectivity options, such as FAX/modem, ISDN, wired/wireless LAN/WAN, etc., may be available at the location where the user is working. In addition, the higher power requirements and large form factor of standard desktop bus extension peripherals are often inappropriate for mobile machines. A new bus standard for low power, small form factor (credit-card sized) hardware devices, called PCMCIA, has been designed specifically for portable computers [PCMCIA93]. Support in the Unix operating system for PCMCIA is needed to make PCMCIA devices available for Unix nomads. In addition, since many of the connectivity options open to nomadic users are likely be serial lines with relatively low bandwidth and high latency, transparent access to an efficient serial line protocol is required.

Perhaps the canonical application for mobile users is electronic mail. A variety of new connectivity technologies, including mobile cellular and packet radio networks, are enabling wide area access to electronic mail beyond the desktop. A critical factor in providing mobile users with electronic mail service is the communication protocols used between the client and servers. Currently, they are optimized for high bandwidth, low latency connections. Most of the newer, wide area connectivity options are low bandwidth, high latency, with a relatively high cost per packet. New application level protocols are required to deal with the different characteristics of these connections.

This paper presents an overview of changes in various Solaris 2.x system components necessary to support nomadic computing. We discuss adding support for power management, kernel and application checkpointing, PCMCIA, serial line connectivity, and mobile electronic mail. The next section describes the power management framework. Section 3. discusses the design of checkpointing, and how it interacts with power management to provide the user with "instant on" capability. In Section 4. , we discuss the implementation of PCMCIA on SunOS 5.x. Section 5. provides an overview of the serial connectivity and the asynchronous serial line link manager. In Section 6. , we discuss electronic mail as an example of what changes may be required in applications to support mobile users. Finally, Section 7. summarizes the paper.

2. The Power Management Framework

Limited power availability is the primary constraint on nomadic computer operation. Since a portable computer consists of a variety of hardware devices, conservation of power can be achieved by matching the power consumed to the activity level of each device. The objective of the power management framework is to match the power consumed by a device to its activity level. In its simplest form, the power management framework can turn devices off when they are idle and turn them on again when they are needed. If the device has special power management hardware, the power management framework can match the power consumption of the device to its activity level in a more fine-grained manner. The result is increased battery life, and therefore increased operating time.

The goals of the power management design are the following:

- Provide a generic, portable power management framework within the SVR4 system design,
- Operate transparently to the user, but include user tunable parameters,

In pursuit of these goals, the framework is divided into three basic parts:

- The device driver mechanisms to support power management,
- The pseudo device driver, `pm-driver`, providing supervisory kernel mechanisms and policies,
- The DDI/DKI [Sun92] changes required to support the framework.

---

4. MS-DOS is a trademark of Microsoft, Inc.

Figure 1 contains a diagram of the power management framework, showing the architectural relationship between the parts, and the parts themselves are described in a separate subsection. In the figure, a light grey arrow indicates that the function is optional, a solid arrow indicates it is required, and a striped arrow indicates the functionality is provided by an implementation dependent mechanism.

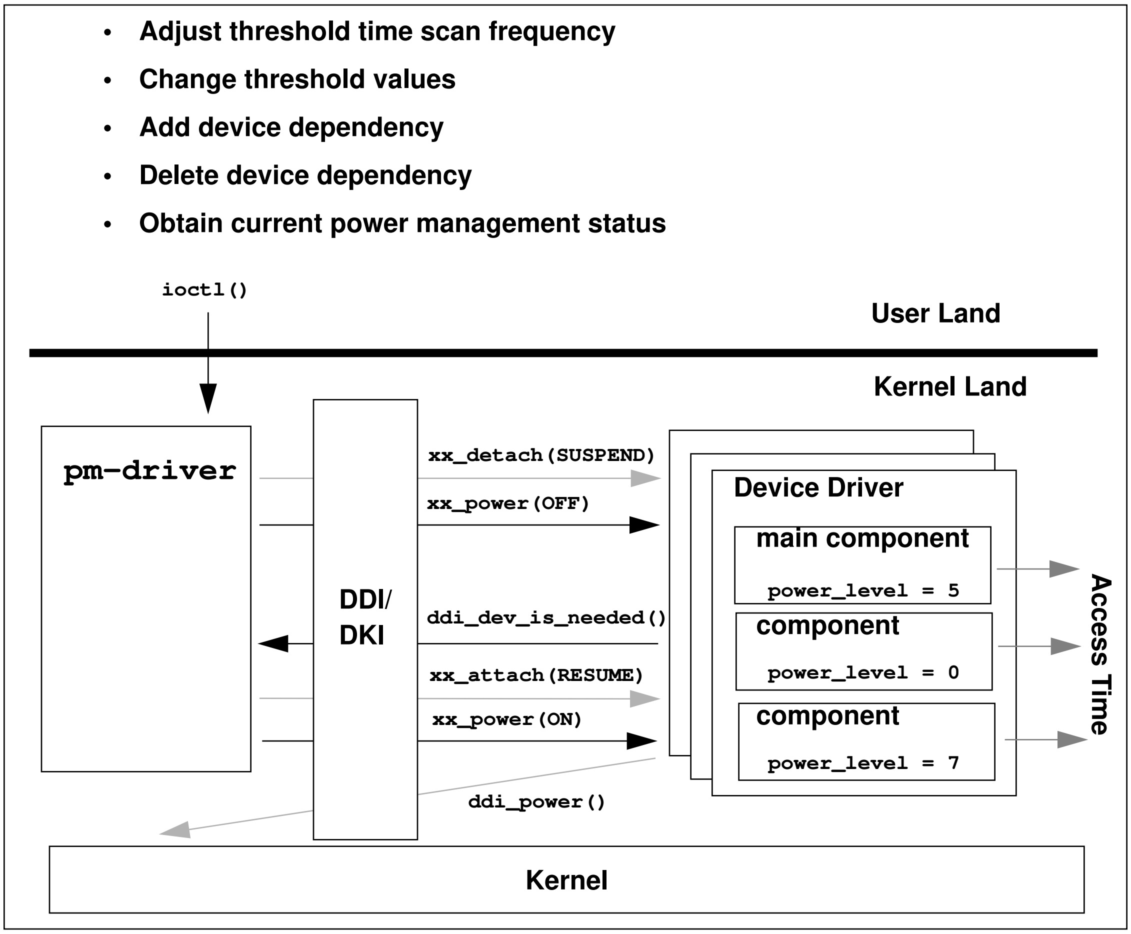

Figure 1 Power Management Framework

2.1 Device Driver Mechanisms

Device drivers are really the key to power management. Device drivers control when a device is physically accessed, whether a device can be suspended, and whether any special power management hardware components are available to the kernel. The power management framework depends on the device drivers to manage the individual devices they control, which is the bulk of the power management. To remain compatible with existing drivers, the framework assumes that it should not try to power manage a device if the driver provides no power management.

The design of the power management framework required adding a new state to device drivers, and rethinking how device driver software relates to the hardware it controls. The new state is device *suspension*. Suspension involves bringing the device to a state where there is no activity, although the power is still on. From a software point of view, suspending the device requires saving hardware registers and driver state to memory. Device suspension is clearly separated from power off, and new driver entry points were added for both suspension and power off. Powering a device off usually involves a writing specific value to a hardware register or issuing a device-specific command to the device.

From the power management point of view, hardware devices can consist of a variety of parts having differing power management capabilities. The extreme example is the entire computer itself, which consists of disk, CPU, display, etc. The power management frameworks models this structure by specifying that a device consists of a number of power manageable hardware units, called *components*. Each component is separately power controllable and has its own set of power parameters. One component is defined to be the main component and is logically where the state of the device is kept. The main component requires the driver to be suspended to save the hardware state before it can be turned off. The driver must also be resumed to restore state when the main component is turned on. The other components must be stateless and do not require the driver to be suspended or resumed to be power managed.

Components collect together state and functionality relevant for power managing a device. Each component has a power level associated with it. A zero power level indicates that the device is turned off, while a device at a nonzero power level is on. The component is responsible for mapping between power states and integer power levels provided by clients to control power. Each component has two basic functions:

- It must export its last access time. A component indicates that it is busy by exporting a zero access time, allowing the framework to detect when the component does not need power management.
- It must notify the power management framework before it is used, if power has been reduced below the level needed for the operation. The component need not notify the framework if the power level is sufficient for the operation.

In some cases, power management within a driver could conflict with efficient device access. The granularity of power control is entirely up to the device driver writer. Each component of a device may be tuned differently, depending on where in the driver the timestamps are updated. Tuning of a component can assure that critical paths within drivers avoid any overhead at the possible expense of some power management granularity. The framework allows the driver writer to make the necessary trade-offs between device access efficiency and power manageability.

2.2 Power Management Pseudo Driver

The power management pseudo driver `pm-driver` is responsible for managing power reduction in the system. The `pm-driver` periodically scans all devices, querying each power manageable device about its usage, and informing the device to shut off if the device has been idle longer than a set amount of time. Additionally, a power manager interface is provided through `ioctl()` operations within the driver. The `ioctl()` operations allow the user to tune the parameters of the `pm-driver`'s power management policies. The `pm-driver` does not power manage any devices not exporting the correct power management information. The `pm-driver` initiates powering down of a component due to inactivity, and powering up of a component through a request from the device driver. If a driver is in a state such that it cannot save all the necessary information, it may return failure. In that case, the `pm-driver` does not turn the main component off immediately, but it may try again later.

Through `pm-driver`, the power management framework recognizes two types of dependencies between devices: logical and physical. A power manageable device has a physical dependency if it has one or more devices attached below it on a set of interconnected buses. A device has a logical dependency if no physical interconnection exists, but a power management constraint exists nevertheless. For example, a graphics card driver that relies on keyboard and mouse inactivity to determine if the display can be turned off has a logical dependency. Prior to power removal, all dependent devices must be properly suspended and powered down before the dependee can be shut down. Physical dependency between devices is implicitly defined in the kernel device tree. The `pm-driver` must be informed of logical dependencies either through a driver configuration file or through an `ioctl()` call. If either a physical or logical dependency exists, then no component of the device is powered down unless the dependency is satisfied.

In order to decide if a device component is idle, the `pm-driver` obtains the time that the component was last accessed, which has been exported by the component. If the last access time is zero, the device is busy. If the difference between the last access time and the current time exceeds the management policy threshold time for the component, the component is deemed idle. If all logical dependents of the device are also idle, the `pm-driver` attempts to turn off the device. If an error occurs, the power down fails and the component is left on.

The `pm-driver` provides all kernel mechanisms and implements power management policy. It is accessed by the user through `ioctl()` operations. These operations allow the user to adjust the frequency of device scans, change component threshold values, or add and delete dependencies, and to obtain the current power management status of a device or of the `pm-driver` itself. The `pm-driver` only controls devices that have been configured (i.e. have at least their threshold time(s) set). Typically, at boot time, a script runs a configuration program that reads a system file and configures all power manageable devices.

2.3 DDI / DKI Changes

As shown in Figure 1, two standard operations in the DDI/DKI [Sun92] have been changed to recognize two additional command parameters and three operations have been added experimentally to the DDI/DKI. The changes allow drivers to interact indirectly with the `pm-driver`, so different power management pseudo drivers can be provided for different hardware platforms, or no pseudo driver if power management is not desired. The changes also permit the implementation of other platform specific power management modules to provide ad-hoc power control for devices which were not initially designed with power control.

The standard DDI/DKI operations `xx_detach()` and `xx_resume()` have two additional command parameters added to their command set, SUSPEND and RESUME. The SUSPEND command parameter requests that the driver save its state, block any further IO, and allow all outstanding IO requests to complete. When it returns, there should be no outstanding requests for IO so that nothing is lost when the device is power cycled. The RESUME command parameter restores the driver's state for a power on. As indicated by the grey arrows in the figure, recognition of these commands by the driver is optional.

`ddi_dev_is_needed()` is a new operation called by the device driver when a component of a device is needed at a certain power level. It returns when the component is at the requested level. `xx_power()` is a new operation that is implemented by the device driver. It takes one command parameter, indicating whether the power should be turned on (ON) or off (OFF), and is called by `pm_driver` to request that the device driver turn the power on or off. `ddi_power()` is a new operation that is called by a device driver which doesn't want to handle power management itself to request that the kernel handle power management with the system default. The black arrows in Figure 1 indicate that the driver is required to implement these.

3. System State Checkpoint and Resume

A major component of the nomadic enhancements is system checkpoint/resume (CPR). A system *checkpoint* saves the state of the entire system, including user-level processes, to nonvolatile storage. A system *resume* restores the system to the checkpointed state at a later time. The primary benefit of CPR for nomadic computing is its role in power conservation. Automatic checkpointing when the battery is low protects the user from undesirable state loss. CPR also matches the people's expectations about how a nomadic machine should be used. With CPR, a machine can be moved from one place to another without going through the time-consuming bootstrap process. Resume times are typically on the order of a minute or less.

CPR has been available in MS-DOS and on a few portable Unix machines running SunOS 4.1 for some time. However, the SVR4 system is considerably more complex than MS-DOS or SunOS 4.1, so the design issues involved are somewhat more challenging. The design goals for CPR in SunOS 5.x are the following:

- CPR should provide extremely reliable service,
- System administration and maintenance should be as simple as possible,

- The design should remain flexible and extensible so it will easily work with new releases of SunOS 5.x,
- The checkpoint and resume times should be as short as possible, to encourage people to use the facility.

There are two possible design strategies for CPR:

- Forcefully shut off all subsystems and take a snapshot of all kernel and user-level process memory images,
- Ask each subsystem to stop and allow the system to save itself.

Most existing implementations of CPR take the first approach [Tadpole93]. The first approach is faster but the outcome is less deterministic. For example, an Ethernet controller might be right in the middle of accepting a packet, and forcing it to shut off causes that packet to drop. For this reason, our design uses the second approach. Figure 2 schematically illustrates the checkpoint process, and the next two subsections examine CPR in more detail.

3.1 System Checkpointing

User processes are the first system component to be checkpointed. When a checkpoint is initiated, all user processes are requested to stop. This is accomplished by sending a new user signal, `SIGFREEZE`, to each process. The default action of `SIGFREEZE` is to stop the process. Applications that need a chance to perform certain housekeeping tasks prior to the checkpoint (e.g. close down network connections, notify servers or clients of impending unavailability) can catch the signal. Once the user processes are frozen, all modified pages with backing store are paged out, and all such pages are invalidated and purged from the page cache. User-level processes are restored from these pages on resume.

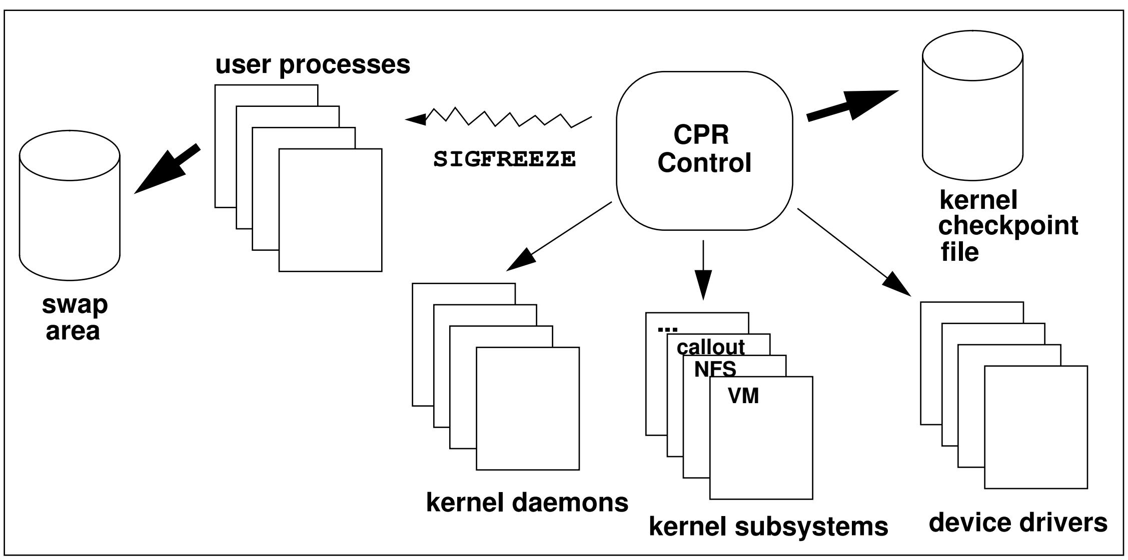

Figure 2 System Checkpoint Process

After all user processes are stopped and their pages are pushed out to the swap device, the kernel daemons are requested to stop. Since each daemon is unique, it needs to stop at a different location. A callback mechanism allows each daemon to specify where it should be stopped. A callback handler can be registered with the CPR callback mechanism by the daemon at thread initialization time if the daemon requires special action prior to checkpointing. The installed handler is part of the daemon and uses the same synchronization mechanisms to stop the daemon.

After all the user processes and kernel daemons are stopped, no further base level activity is possible. The next step is to give all interested subsystems (e. g. VM, callout processing) a chance to prepare themselves for checkpoint. This is accomplished via the same callback mechanism used for stopping the kernel daemons. Only those subsystems that require special processing prior to checkpointing need register callbacks.

Finally, all device drivers are suspended. Although theoretically it would have been possible to design checkpointing so that IO in progress need not complete, the probability that a restarted IO request would succeed on resume is rather low. Therefore, the checkpoint subsystem blocks all new requests and the kernel waits for all outstanding IO requests to complete. After the IO queue is empty, the driver saves the device hardware and software state to data structures in memory using the same suspension mechanisms as power management suspension. Once all the device drivers are stopped, interrupts are turned off and the system enters a completely dormant state. All valid kernel pages are written out to the kernel state file on the root device (e.g. disk or network) and the checkpoint process is complete.

3.2 System Resume

On the resume side, the kernel loading program started by the PROM detects a resumable system by an implementation dependent mechanism and loads in a special resume module. This resume module reads the kernel state file, and restores the kernel memory image and the MMU back to the checkpointed state. Control is then transferred back to the kernel, and the reverse of the checkpoint sequence is executed. When user level processes are ready to run, a `SIGTHAW` signal is sent, in case they need to initialize before beginning normal execution.

4. PCMCIA on Unix

Unlike desktop machines, portable computers have a limited amount of cabinet space for expansion slots and a limited amount of power for peripheral devices. In response to the need for a smaller form factor and a lower power bus, portable computer manufacturers formed the Portable Computer Manufacturers' Card Interface Association (PCMCIA), and developed a new bus standard, having the same name [PCMCIA93]. The PCMCIA standard defines the physical form factor, electrical bus interface (8 or 16 bit), and a software API for small, credit-card-sized devices. The standard allows mass storage, IO and memory peripherals to be used on multiple different hardware system architectures and operating systems. Although the software interface in the standard was developed specifically with implementation on the Intel x86 architecture and the MS-DOS operating system in mind, any computer which implements a bus interface conforming to the PCMCIA mechanical and electrical specifications can utilize a PCMCIA card. Since the existence of the standard is catalyzing volume manufacture, implementing software support for PCMCIA on Unix would provide the Unix user with access to a wide variety of previously unavailable low power peripherals.

The goals of the PCMCIA implementation are the following:

- Provide a software architecture and API that adheres closely to the PCMCIA standard without violating the architectural assumptions of the SVR4 device driver architecture,
- Supply enhancements in areas where the standard is currently underspecified or lacking,
- Assure that the implementation supports maximum portability of PCMCIA card drivers between SunOS on different hardware platforms (e.g. SPARC and Intel x86),
- Feed the results back into the PCMCIA standards process to grow the current software interface beyond its focus on MS-DOS.

In the following subsections, we describe the implementation of the PCMCIA 2.01 standard API on SunOS 5.x.

4.1 The Card and Socket Services Design in Unix

The PCMCIA standard specifies the *Card Services API* as an interface between a device driver for a card and the system software layer managing the PCMCIA bus controller hardware, or *adapter*.

The adaptor provides the physical slot, or *socket*, for the PCMCIA card and a hardware interface between the PCMCIA bus and the system bus. The Card Services API allows a card device driver to control the adapter independently from the specific adapter hardware. The *Socket Services API* is a hardware-independent interface between Card Services and the software layer that controls a particular adapter. It is possible to have multiple Socket Services layers to support multiple different types of adapter hardware.

Figure 3 illustrates the PCMCIA software driver architecture in SunOS 5.x and its mapping onto the hardware. In the figure, grey rectangles are hardware modules and grey arrows indicate interaction between software components and hardware. Black arrows indicate interaction between software components. In the SVR4 architecture, device drivers for bus controllers such as a PCMCIA adaptor are called *nexus drivers* [Sun92]. Drivers for devices on the bus are children of the nexus. Figure 3 indicates parent-child relationships with a striped line. The Socket Services layer corresponds to a nexus driver in the SVR4 architecture, with a separate nexus driver required for each different type of adapter hardware. A particular PCMCIA nexus driver is the parent of all card drivers for cards in that adapter's sockets, and, in turn, the PCMCIA nexus is the child of the system bus nexus. Unlike other device drivers, a PCMCIA card driver does not interact directly with its parent nexus. Rather, as specified by the standard, the card driver calls into the Card Services layer, and the Card Services layer deals with the nexus.

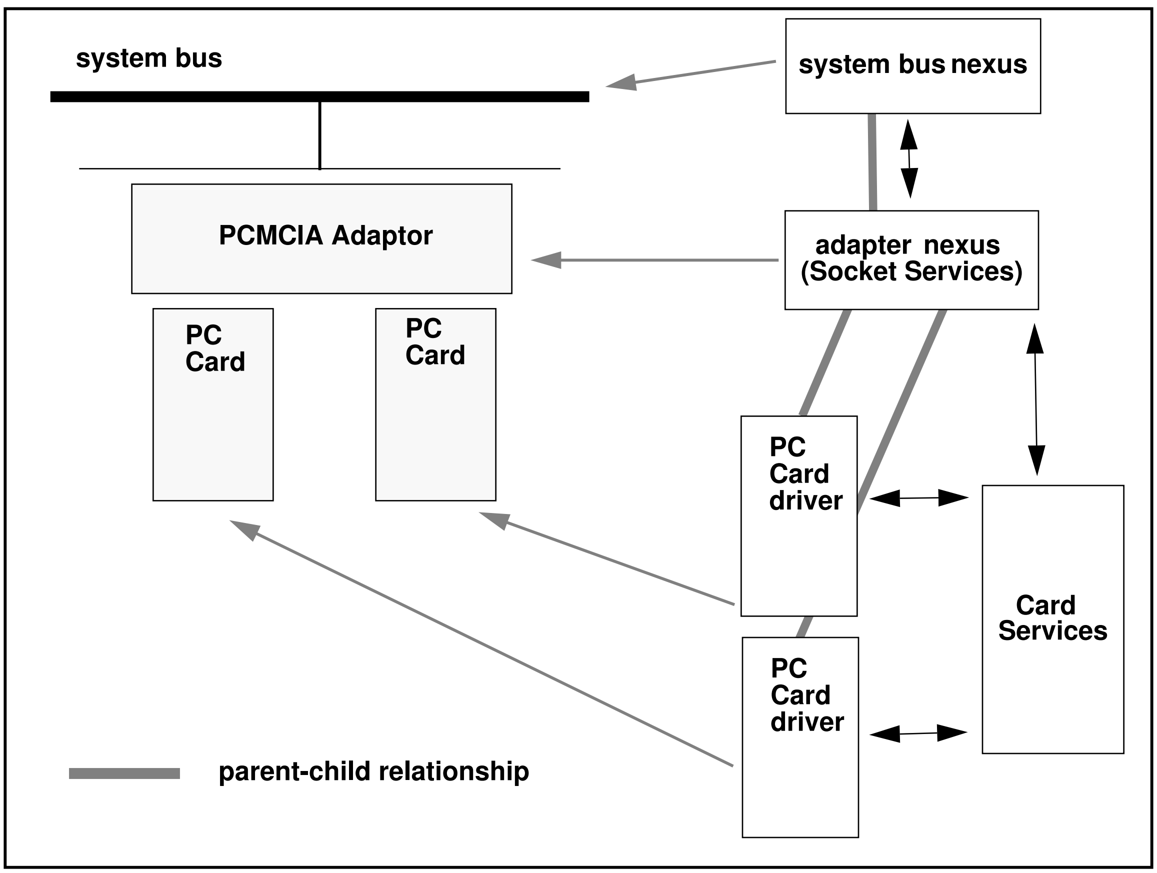

Figure 3 PCMCIA Hardware and Driver Software Architecture

The PCMCIA standard also specifies calls in the Card Services API allowing a device driver to load and unload a Socket Services implementation. The mapping of Socket Services to a parent nexus driver in SunOS 5.x means that this feature cannot be provided on the SunOS 5.x implementation of PCMCIA. Allowing a child to replace its parent nexus is not permitted in the SVR4 device driver architecture.

The SunOS 5.x implementation of Card Services is a collection of function entry points in the kernel. These functions are located in a miscellaneous kernel module that is not part of the device driver hierarchy. The module handles all Card Services requests from all PCMCIA device drivers and makes the calls into the appropriate parent nexus. The Card Services functions require the driver to pass a client handle that uniquely identifies the requesting driver instance. SunOS 5.x Card Services implements this client handle as a pointer to a structure containing the driver's per-instance information. Card Services uses this handle to locate the appropriate parent nexus in an implementation specific way. In the PCMCIA standard, the interface between the parent and the child is publicly specified in the Socket Services interface. In the SunOS 5.x implementation of PCMCIA, this interface is private and not available for use by the device driver itself. However, the interface is available for developers of PCMCIA bus adapter hardware who need to develop a custom nexus.

4.2 Modifications to the Card Services Interface

The specification of the current PCMCIA Card Services API is highly tuned for the MS-DOS operating system. The standard specifies that Card Services has a single function entry point through which all Card Services requests are vectored. This single entry point is specified to require the same number of arguments no matter which Card Services function is being requested. The arguments specified in the standard are of fixed type, and different type arguments for different Card Services calls must be packed as untyped bytes into a variable length byte array. In the current PCMCIA standard, each processor type, processor mode (protected, real, etc...), subroutine call method (near, far) and operating system on which Card Services is implemented specifies an implementation-specific binding that dictates the details required to make the call. The model is very similar to an MS-DOS INT 21 function call.

This design hinders driver portability for operating systems, such as SunOS 5.x, that run on multiple system architectures and multiple processors. As part of Sun's efforts in the PCMCIA standard process, a C language calling convention is being proposed. The proposal implements a single Card Services function entry point with a variable argument list of pointers to function-specific structures and/or opaque data types, somewhat like an `ioctl()`. Details such as the sizes of various address and data types, byte ordering for shared structure members, and generic system resources managed by the kernel are kept hidden from the driver developer. Selective data hiding allows a device driver to be source compatible between SunOS 5.x on the SPARC architecture and SunOS 5.x on the Intel x86 architecture. Certain differences between the SPARC and Intel x86 architectures, such as card IO port accesses, are hidden using register access macros supplied by the PCMCIA header files. On SPARC, card IO registers are mapped into the device driver's virtual memory address space, while on the Intel x86, the registers are accessed using privileged I/O instructions, but the macros hide this detail from the driver writer.

4.3 The Card Information Structure (CIS) Interface

One of the most important features of the PCMCIA standard is the specification of a mechanism making every PCMCIA card selfidentifying. Every PCMCIA card has an area in which card information is stored, called the *Card Information Structure* (CIS). The CIS is a singly-linked-list of variable-length tuples. Each tuple has a one byte code describing the tuple type, and a one byte link which is the offset to the start of the next tuple in the list. The CIS provides a wide variety of information about the PC card. Each tuple can contain subtuples that elaborate on the information provided by the parent tuple. Although the PCMCIA standard originally only specified enough bits for 256 tuples, as the standard grew, the need for more arose. Since no standard extension mechanism has been proposed, extensions have arisen in an ad-hoc manner.

The only method specified by the PCMCIA standard for dealing with CIS structures is a set of Card Services functions that return the raw contents of a tuple. The device driver is required to take care of tuple parsing itself, even for tuples whose structure is fixed by the standard. The SunOS 5.x Card Services implementation provides the standard Card Services functions, however, it also includes a tuple parser that parses all tuples specified by the standard into structures defined in the PCMCIA header files. A device driver need not include any tuple parsing code itself unless information from a nonstandard tuple is required.

PCMCIA CIS structures are also used by the PROM during booting to build the kernel device tree on SPARC systems. SPARC systems contain a rudimentary CIS interpreter and tuple parser in the body of the boot PROM. This CIS interpreter knows about certain common tuples that provide device identification and configuration information, and the interpreter and parser build simple nodes in the kernel device tree for each card found at boot time. In addition, card developers can include Fcode, the programming language used by Sun's boot PROM for selfidentifying devices, in the CIS of their cards, allowing the boot PROM to configure PCMCIA cards just as Sbus cards are configured currently. For cards that appear to the system as mass storage or network devices, the PROM can load boot images or establish network connections, allowing the PROM to boot the SunOS 5.x kernel from files read off PCMCIA mass storage or network devices. In SunOS 5.x for Intel x86 systems, using a PCMCIA card as a boot device is not yet supported, since the boot PROM on Intel x86 machines is not programmable.

5. Serial Wide-Area Connectivity and Link Management

Traditionally on the desktop, the primary connectivity option available to users has been 10 Mbps Ethernet. The most common form of network connectivity available to nomadic users currently is a simple serial modem connection (2400 to 14.4K bps, faster with compression) via an analog telephone line. More recent innovations in mobile connectivity have made alternate options available, such as spread-spectrum and packet radio, cellular, and satellite. One characteristic common to most nomadic connectivity options regardless of their bandwidth and latency characteristics is point to point serial transfer. Support for efficient serial connectivity is therefore essential for nomadic users.

The goals of the serial connectivity design are the following:

- The serial connectivity protocol should slip easily in at the ISO data link layer, and use standard higher level and lower level interfaces,
- Performance of the protocol over low bandwidth, high latency serial networks, such as analog telephone, should be adequate,
- The serial protocol should be accommodated by existing network infrastructure (e.g. routers),
- Connection management should optimize connection time, so that expensive or unreliable connections need not remain active when they are not in use,
- The protocol should contain proper support for security.

These goals lead to the incorporation of the Point to Point Protocol (PPP) [Simpson92a] [Simpson92b] [McGregor92] [Lloyd92] into SunOS 5.x, and the development of the asynchronous link manager. The next two subsections describe these new system components.

5.1 Point to Point Protocol (PPP)

Most network applications built for the desktop environment have been engineered for the Ethernet, and specifically for the TCP/IP or UDP/IP protocols. For nomadic users to be able to employ standard network applications, like the Network File System (NFS) over a serial line, the same kind of higher level services available on Ethernet, e.g. TCP/IP and UDP/IP, are required over a serial connection. PPP is a standard protocol developed by the Internet Engineering Task Force Network Working Group for point-to-point links such as dial-up modem servers [Simpson92a] [Simpson92b] [McGregor92] [Lloyd92]. With PPP, applications can access the wide-area serial connection through the same system interfaces and higher level protocols used for the local area network.

The PPP protocol consists of three components:

- An encapsulation method for datagrams over serial links,
- A Link Control Protocol (LCP) for the establishment and configuration phases of starting a connection, and for testing,

- A family of Network Control Protocols (NCP's) for connecting over different network layer protocols.

The PPP design specifies an encapsulation protocol over bit-oriented synchronous links and asynchronous links with 8 bits of data and no parity. The links must be full duplex, but can be either dedicated or circuit-switched. An escape mechanism is specified to allow control data such as software flow control (XON/XOFF) to be sent transparently, and to remove control data interjected by intervening software or hardware. Only 8 additional octets are necessary for encapsulation, and the encapsulation can be shortened to 2 octets if bandwidth is at a premium, supporting lower bandwidth connections. The encapsulation scheme provides for multiplexing different network-layer protocols over the link at the same time. As a result, PPP can provide a common solution for easy connection with a wide variety of existing bridges and routers. Finally, PPP provides two protocols for authentication: a password authentication protocol and a challenge-handshake authentication protocol.

5.2 The Link Manager

The asynchronous serial line link manager is a user level daemon that takes care of maintaining serial line connections. It automates the process of connecting to a remote host when PPP service is established. The connection can be initiated either by simply sending a IP datagram to a disconnected host, or by receiving notification from a remote host that a connection is desired. In this way, network services such as NFS can be provided without requiring a continuous, physical connection. In Figure 4, the position of the link manager in an experimental serial line architecture is shown. The link manager is normally started at boot time by `init`. It reads a configuration file, builds internal data structures, opens up a connection to the kernel IP-Dialup module (`ipdcm` in the figure), creates and opens the FIFO on which it listens for incoming (dial-in) requests, and then enters an idle state waiting until a message appears on either of the opened connections.

An outbound connection is initiated when the link manager receives a connection request message from IP-Dialup. It checks to see if the connection request corresponds to a configured "path", consults the UUCP data base files for modem and destination system information, and then places a phone call to the destination host. After the physical link is established (i.e. the two hosts are connected over the phone line), the link manager configures and initiates PPP. Once the data link layer is established and the PPP modules on the peer hosts are communicating with each other, the link manager starts IP and, if specified, modifies the local route table to indicate that the newly created path is to be used as the default route. The link manager then passively monitors the connection until an event such as idle time out, line disconnect, or an error condition occurs. When this happens, it will disconnect from the peer, clean up external data structures like the route table, and return to the idle state.

An inbound connection is initiated when the link manager receives a notification that the login service has opened the named pipe FIFO. The login service is a separate process that is invoked by the `login` program. When the login service is invoked, it opens the named pipe FIFO and passes the login name and the file descriptor for its standard input to the link manager listening on the other end of the named pipe. It then waits until the pipe between it and the link manager is closed, indicating a disconnect, then exits. This allows `login` to regain control over the serial port again and display the login prompts. When the link manager is notified that the login service has connected to the named pipe FIFO, it reads the information written to the pipe, checks to see if the login name corresponds to a configured path, and then configures and initiates PPP. The rest of the scenario is the same as that described for outbound connections.

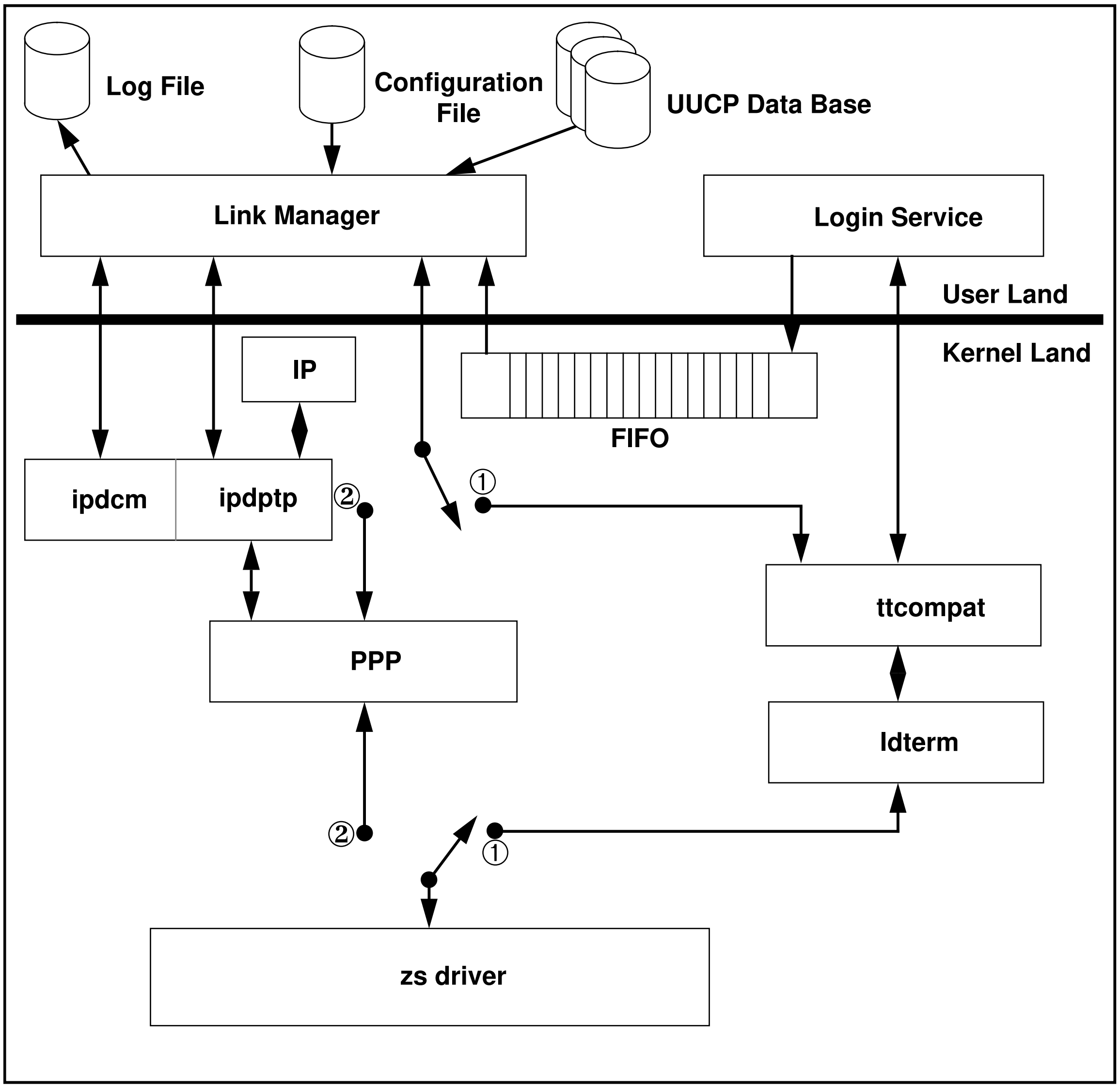

Figure 4 Components and Interfaces in the Link Manager

6. Nomadic Electronic Mail

Electronic mail is the canonical example of an attractive mobile application. The asynchronous nature of electronic mail allows a mobile user to send email from one location and get the reply at another. Mobile users require the same level of electronic mail service as their deskbound counterparts. In addition, since a wide variety of wide-area and local-area connectivity options of varying cost and performance are likely to be available to a mobile user, nomadic access to electronic mail should reduce the need for connectivity unless it is absolutely required, and allow good access even over high latency, low bandwidth connections. For many nomadic users, remote mail may be the single most convenient way of maintaining electronic connectivity, supplanting terminal emulators and file transfer which are the current most widely used options.

Based on these considerations, the following goals for the nomadic electronic mail application were defined:

- Access to electronic mail should be possible over any wide-area or local-area connection,
- It should be possible to manipulate mail state while disconnected from the network, so a user can read mail without requiring a potentially costly wide-area network connection,

- The mail application should be usable even after an unanticipated disconnection from the network or a server failure,
- A user should be able to indicate that they need a certain level of functionality when they are disconnected and the mail application should accommodate this request,
- The nomadic electronic mail application's user interface should differ from the standard desktop UI only in those areas where specific nomadic problems must be addressed.

The following two subsections describe design solutions for these goals.

6.1 IMAP Mail Protocol

The underlying mail protocol used by the electronic mail system plays an important part in determining the connectivity characteristics of the electronic mail application. A variety of currently popular mail protocols and other ways of remotely viewing mail (POP [Rose91], PCMail [Lambert88], remote X [Scheifler92]) were examined and the Interactive Mail Access Protocol (IMAP) [Crispin90] [Crispin92] was selected. The primary reason for selecting IMAP is because it was designed as a interactive protocol rather than a protocol for simply moving messages from one message store to another. The IMAP protocol makes efficient use of the underlying connection bandwidth by breaking the message into logical chunks, allowing the mail reader to request only the parts of the mail message that the user really wishes to see. For example, if a mail message is in a multimedia format such as MIME, an IMAP-based mail reader could only request the text of the mail message and allow the user to make a choice as to whether or not they wanted to wait for a multimedia attachment to be downloaded.

Although IMAP was a good starting point, its main use to date has been over standard Ethernet and thus its implementation and definition required improvement to support mobility. For example, IMAP sends an envelope structure over the connection for each message, even though the envelope structure is unneeded unless the user actually indicates a desire to reply. To increase the efficiency of IMAP over low bandwidth, high latency wide-area connections, the protocol was modified to reduce the amount of such unnecessary data sent across the connection. In addition, IMAP originally did not support disconnection. Disconnected operation is handled by calculating a checksum on the server to determine if some other mail reader has modified the section of the remotely cached file. Although the checksum should be updated every time an operation affecting mail state is executed, transmitting the checksum separately could cause much higher packet traffic. By piggybacking the checksum on the completion reply packet, the update information is sent as part of the standard request/reply protocol, eliminating the need for a separate request/reply to handle the checksum. At the moment, the mobile IMAP does not support full functionality while disconnected, however. For example, full support of searches for text in the body of a message is impossible unless the body of the message is local or the connection is up.

6.2 Nomadic User Interface Enhancements

For the most part, the user interface of a nomadic mail reader application need differ little from one for desktop use. Indeed, close user interface compatibility between the mail reader for nomads and for the deskbound means that users need not learn a new command interface when they take their machine on the road. *ROAM* is a prototype nomadic mail reader similar in basic function to the standard Sun reader *Mailtool* but with a Motif interface. Besides those places where Motif and OpenLook differ, the command set is almost the same as *Mailtool*.

The exceptions are where additional command support for remote or disconnected operation is required. A command interface allows messages to be downloaded for caching on the local machine, so that users can view mail while disconnected. Users may also want to know which messages are local and which are remote, to avoid an unpleasant surprise while disconnected when they attempt to read a message which is not there. Headers of messages that are not locally cached appear differently from those that are only available over the network, and message bodies that are not local are displayed as half full icons. Locally cached messages have their headers displayed in roman font, while remote messages are in italic. Another user command allows the mail queue to be manipulated, so that unsent messages can be deleted if desired. Finally, users who are connected over wide area serial links need the ability to specify more stringent searches to reduce the amount of traffic and number of misses when trying to find a particular message. Detailed search capability was added including search by dates, text in message body, status of messages and many others. Using this search capability, a user could search for a message received this week concerning a certain topic eliminating the large number of messages that would appear if all messages received for this topic were searched. The search is then run on the server, rather than downloading all message bodies to the client and running the search locally.

7. Summary

Fostering a more nomadic usage model in Unix requires changes throughout the entire system, from the kernel to the commands of user applications. The primary and most important change in the kernel is automation of power monitoring and control, to reduce the drain on batteries. Power management increases battery life, allowing users to work longer without having to plug into wall power or change batteries. System checkpoint/resume provides ease of use in such areas as quick start and reliability. With system checkpoint/resume, a mobile user can quickly shut down operation at one physical location and start up at another without requiring a lengthy initialization.

The new PCMCIA standard presents an opportunity for Unix-based machines to leverage off of a wide variety of new communication, IO, and storage devices. Implementing the standard in Unix has required a variety of changes, since PCMCIA was originally developed for MS-DOS. The changes map concepts such as Socket Services onto components of the existing SVR4 device system design. Mostly, however, the PCMCIA Card Services layer has been maintained intact, providing high fidelity to the standard in the part of the API which matters most to device driver writers.

Serial line connectivity is and will remain the primary means for nomadic users to tap into the Internet. The PPP protocol, developed by the Internet Engineering Task Force, is an efficient serial line protocol allowing connection with existing network infrastructure. Since serial connections are generally more expensive and less reliable than Ethernet, connection management is required. The asynchronous serial line link manager hides the details of making serial connections, so that a serial connection looks exactly like a standard TCP/IP connection to an application.

While many existing network applications can be used unchanged in the nomadic environment, others may require modification. Electronic mail is an example of the canonical nomadic application, since email is so useful to nomadic users. The IMAP interactive mail protocol, modified for nomadic use, provides the basis for efficient support of email over a serial line and for disconnected operation. The changes made to IMAP are a useful prototype for how application-specific protocols can be changed to support a nomadic model. In addition, modifications to the user interface allow users to specify caching upon disconnection, and provide a variety of other command features specific to the nomadic environment.

8. References

[Crispin90] Crispin, M. “Interactive Mail Access Protocol - Version 2,” Internet Engineering Task Force RFC 1176, 1990.

[Crispin92] Crispin, M., “IMAP2BIS - Extensions to the IMAP2 Protocol,” Internet Engineering Task Force RFC 1176, 1992.

[Lambert88] Lambert, M., “PCMail,” Internet Engineering Task Force RFC 1056, 1988.

[Lloyd92] Lloyd, B., and Simpson, W., “PPP Authentication Protocols,” Internet Engineering Task Force RFC 1334, 1992.

[McGregor92] McGregor, G., “The PPP Internet Control Protocol (IPCP),” Internet Engineering Task Force RFC 1332, 1992.

[PCMCIA93] *PCMCIA Standard Release 2.01*, Portable Computer Manufacturers’ Card Interface Association, 1993.

[Rose91] Rose, M., “Post Office Protocol - Version 3,” Internet Engineering Task Force RFC 1081, 1991.

[Scheifler92] Scheifler, R.W., and Gettys, J., *X Window System*, Digital Press, 1992.

[Simpson92a] Simpson, W., “The Point-to-Point Protocol (PPP) for the Transmission of Multi-protocol Datagrams over Point-to-Point Links,” Internet Engineering Task Force RFC 1331, 1992.

[Simpson92b] Simpson, W., “PPP Link Quality Monitoring,” Internet Engineering Task Force RFC 1333, 1992.

[Sun92] *SunOS 5.1 Writing Device Drivers*, SunSoft, Part No. 801-2871-10, 1992.

[Tadpole93] “The Nomadic Computing Environment”, Tadpole Corp., 1993.