To appear in Proceedings of the 1994 IEEE Computer Conference, San Francisco. Copyright © 1994 IEEE

The Newton Operating System

Robert Welland†, Greg Seitz†, Lieh-Wuu Wang,
Landon Dyer, Tim Harrington, and Daniel Culbert

Apple Computer Inc.
†Wayfarer Communications Inc.

Abstract

The Newton MessagePad Personal Digital Assistant (PDA) is the first in a class of devices distinguished by their pen-based user interface, communications capability, small size, and low cost. A PDA operating system needs to support the simultaneous demands of user interface, applications, and communications. It must operate in an environment that has little main memory, no secondary storage, and a small power source. We describe an operating system designed to address these needs. It includes a micro-kernel, a memory management subsystem, and support for removable storage and I/O devices.

1: Introduction

The Newton™ MessagePad™ is the first in a line of Personal Digital Assistants (PDAs). These pen-based devices are small, lightweight, and battery powered. The hardware is constrained by these requirements. Table 1 summarizes PDA hardware characteristics.

Table 1. Hardware Characteristics

Characteristic	Why
Small screen and tablet	Cost, power, size, weight
Small, slow main memory	Cost, power
Diskless	Power, cost, size, weight
Modest power source	Weight
Powerful CPU/cache	Performance demands: pen input, handwriting recognition, and dynamic programming language

PDA software must operate in this environment while still providing a responsive user experience and uninterrupted communications.

The Newton MessagePad has the characteristics detailed in table 2. Severe RAM constraints and the absence of a permanent secondary storage device present significant design challenges. The RAM should be allocated on an as-needed basis. Software should adjust dynamically to usage patterns.

Table 2. MessagePad Characteristics

Characteristic	What
RAM/ROM	640 KBytes, 4 MBytes
Power supply	4 AAA cells
Processor	20 MHz ARM 610
Mass storage	Optional PCMCIA memory card

Persistent user data is stored directly in RAM. This data must be carefully guarded against corruption. It is important that user data survive even if the system reboots. It should be possible to store persistent data in a compressed form and decompress it on demand. Caching oft-used data is an important optimization in the presence of expensive compression algorithms (CPU time = power consumption).

A small power source requires careful management. If the processor were allowed to run continuously, the battery life would be measured in minutes rather than days or weeks. In low power systems, the instructions-per-second (IPS) rating of a processor are not as interesting as the processor's IPS-per-watt rating. IPS is interesting for computationally intensive operations like handwriting recognition, but averaged over time the IPS rating of a low power system would be quite low. Software must take full advantage of the power management features of the hardware.

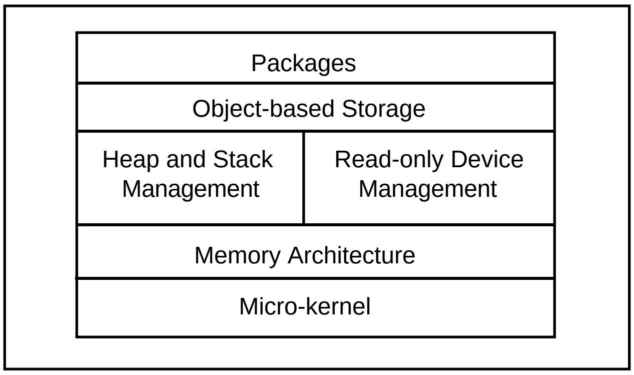
PCMCIA cards allow the system to be enhanced with additional memory or peripherals. These devices are different from more traditional expansion devices because

Newton, MessagePad, and AppleTalk are trademarks of Apple Computer, Inc.

they can be removed at a moment's notice. Software must be able to use these devices while still functioning when they are removed unexpectedly. It is important that these cards be easy to use. Peripheral devices should provide their drivers on-card rather than on removable media (such as floppies). Software must define how drivers are loaded from external devices.

The Newton operating system addresses these issues. Figure 1 shows the basic structure of the operating system.

Figure 1. Structural overview



The micro-kernel provides a multi-tasking framework for higher level services. Multi-tasking makes it possible to modularize concurrent subsystems. It is described in section 2.

The memory architecture is described in section 3. It provides the framework for managing the address space and memory resources. This subsystem provides the basis for managing the scarce RAM resources.

Section 4 details some of the features of the memory architecture implementation.

Section 5 explores some of the specific memory saving measures implemented with the memory architecture.

Section 6 is a brief summary of our persistent storage model.

Finally we describe the means for loading code and data from external sources. The concept of packages provides this ability and is described in section 7.

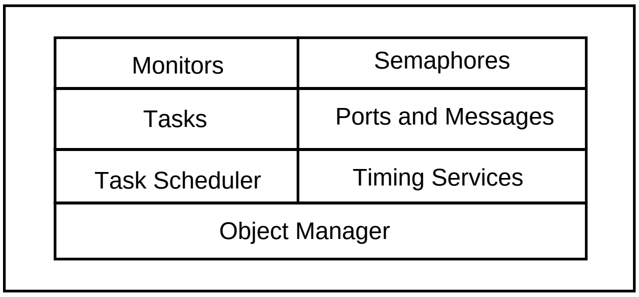
2: The micro-kernel

The micro-kernel provides the core concurrency functionality. Figure 2 shows the basic elements of the micro-kernel.

2.1: Object management

All kernel components exist as objects in the micro-kernel. Kernel objects are managed uniformly by an object manager. Every object is owned by some task. When a task dies, the objects it owns are scavenged by the object manager or can be bequeathed to another task. Kernel objects are represented to clients as object identifiers that are correlated to an object residing in the kernel. The client normally wraps the object identifier with an interface class. Calls on that class pass the object identifier along with arguments to the kernel. Kernel objects are kept separate from their clients for safety. Errant behavior by a client cannot make a kernel object inconsistent (though the object may be rendered useless).

Figure 2. Micro-kernel overview



2.2: Tasks

Tasks provide the basic unit of concurrency. Tasks are lightweight in that they can be context switched quickly. We will see later that tasks are also lightweight in the amount of memory they consume. Tasks contain little more than the processor state needed to save and restore the thread of execution and some additional fields used to manage the blocking state of the task. At task creation time, the creator can provide an initial state for the task. This state is copied onto the task's stack. The location of the object can be accessed by a kernel call. This is used to implement per-task globals. It can also be used to implement an object-oriented model of concurrency.

2.3: Task scheduling

Tasks are scheduled preemptively. We use a strict priority-based scheduler. Because of the highly cooperative nature of task utilization in Newton, strict priority scheduling has been sufficient. Tasks are switched approximately fifty times a second. This number was derived by experiment.

2.4: Ports and messages

Tasks can communicate by sending messages. Messages are sent to and received from ports. Ports are used as the synchronizing mechanism. That is, a task will block on a port if it tries to receive from a port that has no messages pending. Similarly, a sending task will block if there is no receiver waiting. It is possible to send messages asynchronously. This keeps the task from blocking.

The message specifies an address region. To send a message is to give the receiver the right to read the contents of that region. To receive a message is to specify the address region where a message should be placed. Message copying is done by the kernel. We will see later that the memory architecture controls the readability and writeability of address regions. It is therefore important to make sure that the kernel is mindful of these controls during message copying.

Timing services are provided as a part of the port and messaging model. Send and receive calls can specify a limit on the amount of time (timeout) they will block. It is also possible to specify that a message is to be delivered at some point in the future. Time is specified as a 64 bit number that describes absolute time. Delayed messages are particularly useful when writing event-based code.

2.5: Semaphores

Semaphores are provided for high speed task synchronization. A variety of semaphores are provided. The simplest semaphore model uses a shared memory location and only enters the kernel when there is contention. This approach is very fast but can be corrupted by errant memory writes. It also requires that all semaphore clients have write access to the shared semaphore state.

At the other extreme, it is possible to specify a compound semaphore. These semaphores consist of a sequence of semaphore operations that are executed atomically in the kernel. Compound semaphores can be used to create complex locking sequences that are free of "deadlock windows" because they are executed atomically.

2.6: Monitors

A monitor has many of the characteristics of a task except that it executes on demand. A monitor exports an interface that can be called by many clients at once. However, only one request is allowed to execute at a time. Clients wait in line for their turn. It is helpful to think of a monitor as a heavyweight concurrent object: clients see a monitor as an object in the same sense as any other kernel object. Monitors have initial state just like tasks. This can be used to initialize the state of an object associated with the monitor.

Monitors are used to implement kernel extensions. For example, the memory architecture uses monitors to implement managers that are called when a task faults. Many tasks can fault, but only one fault is processed at a time. Note that these kernel extensions are not a part of the kernel. It is in this way that new services and drivers can be added to the kernel without recompilation.

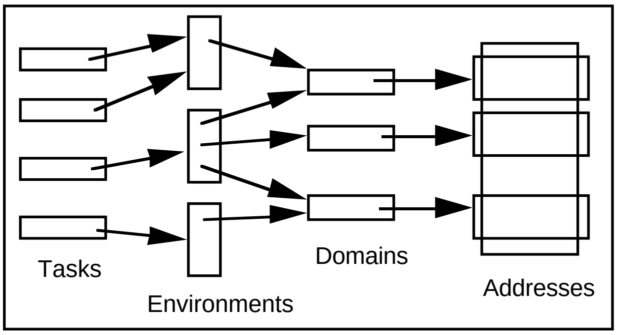
3: The memory architecture

The memory architecture provides the framework for managing address and memory resources. The Newton operating system supports a single address space model. In this model, all tasks reside in a single address space. This approach has a number of advantages. Addresses can be used to uniquely identify objects across all tasks. This facilitates cooperative multi-tasking by allowing tasks to exchange addresses (instead of state) where appropriate. As we will see, it also dramatically reduces the per-task overhead for page table management. In our system, all tasks share a single (multi-level) page table.

The single address space model also eliminates the task/thread duality found in all multiple address space kernels that support lightweight concurrency [1]. In a multiple address space kernel, tasks are normally heavyweight because they must context switch the page tables (and sometimes flush caches). Lightweight threads provide low cost concurrency. Tasks and threads are one and the same in our model.

Most existing single address space models do not provide access protections. This leads to less reliable systems because one task can corrupt the state of another task. Most kernel implementations have provided task protection by supporting multiple address spaces. This is because traditional memory management units have been designed with this approach in mind. As part of our kernel development effort, we have developed a memory management unit optimized to support our single address space model [2].

Figure 3. The memory architecture



The address space is managed by address-oriented services [3]. These services manage regions of the address space by manipulating memory and permission mappings. For example, a virtual memory subsystem can be viewed as an address-oriented service that manages a backing store and working set by responding to client faults. The memory architecture provides a framework for implementing address-oriented services. These services sit on top of the memory architecture and manage stacks, heaps, and read-only data. These services are often invoked when a client task faults. For example, stacks are automatically grown when a task references a stack location (address) that has no storage associated with it. The resulting fault will invoke the stack manager which will in turn map storage at that location and resume the task. We call such services fault-driven because they are invoked by faults. The rest of this section will detail the single address space model. Figure 3 shows the basic elements of our model.

3.1: Domains

The single address space is broken up into a set of regions called domains. Domains are normally contiguous address ranges. Domains may not overlap each other.

A domain is normally used to implement an address-oriented service. Each domain would normally be associated with a specific service. There may be a number of domains providing the same service. For example, there are separate domains for the kernel and client heaps.

Figure 3 shows three domains and their associated address ranges.

3.2: Tasks and environments

An environment consists of a set of domains. In figure 3 there are three environments. The first and third contain single domains. The second contains all three domains shown.

Associated with each task is an environment. This environment describes the set of domains that the task has access to. If the task attempts to access a domain outside its environment, the kernel will generate a domain access exception, which the task can catch.

Access rights do not equate to the ability to read and write. It simply means that the task can try to access that region. The behavior of reads and writes are controlled by an address-oriented service associated with the domain. These services manifest themselves as domain managers. The structure of domain managers will be detailed in the next section.

3.3: Memory management unit support

Our memory management unit (MMU) supports domains and environments directly. Associated with each primary page table entry (each primary entry corresponds to a 1 MByte region of the address space) is a domain number. This number is used to specify which domain owns this range of addresses. The current hardware supports sixteen domains.

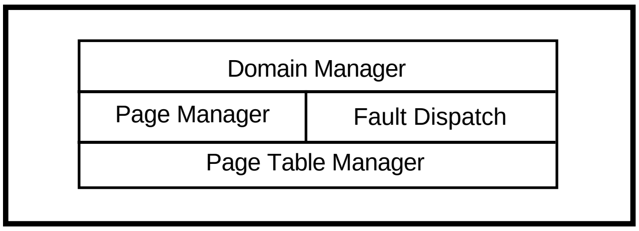
Environments are supported by the domain control register (DCR). For each domain number there is an access control field in the DCR. This field specifies if the currently running task has access rights to the corresponding domain.

For every memory access the domain number associated with the access address is computed and compared with the corresponding DCR field. If the field allows access to the domain, then the memory operation continues. Otherwise, a fault is generated.

4: Memory architecture implementation

Domain managers are used to implement address-oriented services. A domain manager is associated with one or more domains and is responsible for handling faults that occur within that domain. A domain manager can make calls on the kernel-level page management and page table management subsystems. In this way storage can be acquired or released and the page tables can be modified. It is with these resources that a domain manager responds to a client fault. The following sections detail the interactions between these components. Figure 4 details the basic components of the implementation.

Figure 4. Implementation model



4.1: The fault dispatcher

The fault dispatcher receives memory management unit faults and directs them to the appropriate domain manager. A fault occurs when a task tries to access a domain in a manner not presently supported by the mappings. The fault dispatcher blocks the faulting task and queues it with the appropriate domain manager. It is in this way that a domain manager gets invoked.

4.2: The page table manager

The page table manager handles all mapping requests from a domain manager. The page table manager creates and maintains the multi-level page table used by the memory management unit. Mapping operations are either memory or permission mapping operations. A memory mapping operation consists of a physical memory resource (such as a page of RAM or ROM) and an address. A permission mapping operation provides a permission (such as read-only or read-write) and an address region. Operations for removing mappings are also provided. By convention, we designate a page table mapping as

v→(p, perm) where v is a virtual address, p is a physical address, and perm a permission. A permission can specify read-only access (R), read-write access (RW), and no-access (N).

4.3: The page manager

The page manager provides pages of physical memory that can be used by a domain manager. The page manager manages a pool of pages that can be used by any domain manager. The page manager implements a protocol for retrieving pages from domain managers. In this way, pages can be recycled between domain managers.

The page table manager is also a client of the page manager. It calls the page manager when it needs storage for new page tables.

4.4: Domain managers

A domain manager provides the address-oriented service associated with a domain. A domain manager can map physical memory resources into a region of the domain space. It can also manipulate the access permissions. Often these operations are combined.

A domain manager is called whenever a task faults in its domain. The domain manager is provided with the details of the fault (the fault address, the kind of access attempted, and the kind of fault). The domain manager must determine how to respond to the fault. The simplest response is to generate an exception. This would be appropriate in the case that the task has done something wrong (for example, accessing outside the bounds of a heap). In the normal case, the domain manager would attempt to perform some mapping operations (for example, to grow the tasks stack) and restart the task.

In performing its duties, the domain manager will interact with the page manager and page table manager to acquire, and possibly return, physical memory resources and to perform address and permission mappings.

Domain managers are implemented as two monitors and a semaphore. One monitor fields requests from the page manager. The other monitor fields requests from the fault manager. These requests need to be handled by separate monitors to avoid the deadlock case that would result if handling a fault results in a page manager request (monitors only handle a single request at a time). The semaphore is used to serialize monitor access to a common database of mappings. A client level wrapper class is provided that encapsulates the common functionality of a domain manager. Specific domain managers subclass this wrapper and provide service specific behavior and state.

5: Memory management

In this section we will describe two address-oriented services. We also describe a technique, sub-page management, that allows us to more carefully control memory usage.

5.1: Stack and heap management

Task stacks start out with a minimal size and grow automatically. It is the job of the stack manager to grow (and shrink) stacks on demand. The stack manager also allocates address ranges for stacks. By default, the stack manager will allocate each new task a 32 KByte region of address space in the stack domain. Tasks can request larger stack regions if necessary. Initially, there is no storage associated with a task's stack region. As the task uses the region, it faults and the stack manager allocates storage. Guard bands separate stack regions so that attempts to access outside of the region bounds can be detected. If a task generates an access in the guard area an exception will be generated. Because storage is allocated only as needed, the amount allocated is proportional to need.

Unfortunately, most memory management units map memory in rather large chunks (4 KBytes is common). This can be wasteful if a task only needs a few hundred bytes of stack. Using the sub-page management technique described below, the stack manager can grow a task's stack in increments of 1 KByte. This results in a significant savings.

Heap management is broken into two parts. A client level part manages the allocation and deallocation of units of memory. The client level part is also responsible for heap compaction and handle management. The client level part makes calls on a kernel level part to grow and shrink the total amount of memory in use. Because heaps are grown explicitly (i.e., an allocation call is made) the heap manager is not fault driven. When more memory is needed, the client level piece makes a call directly to the kernel piece. This interface is exported by the kernel level heap manager. The kernel level part also allocates and manages address regions for heaps. A heap is normally allocates as a large region (256 KBytes).

In actuality, the kernel level part of the heap manager and the stack manager are one and the same. They differ only in the way address regions are allocated and how growth is managed.

5.2: Read-only data management

There is a significant amount of read-only data in the Newton MessagePad. Executable code, dictionaries, and read-only objects are a few examples. In addition to on-board ROM, PCMCIA cards may be the source of read-only data. Often this data can be compressed, resulting in significant growth in the effective amount of storage available. Finally, PCMCIA cards tend to have slow access times, and there can be significant performance and power gains from caching these storage devices.

The read-only storage manager provides a way of managing read-only data in a manner that incorporates caching and compression. It manages a pool of memory that is used as a cache. Compressed data is decompressed into the cache and made available to clients. The read-only storage manager is similar to virtual memory with support for decompression.

The read-only storage manager takes advantage of sub-page management as described in section 5.3. This makes it possible to manage the cache as a pool of 1 KByte blocks instead of a pool of 4 KByte blocks. This results in a significantly higher cache hit rate.

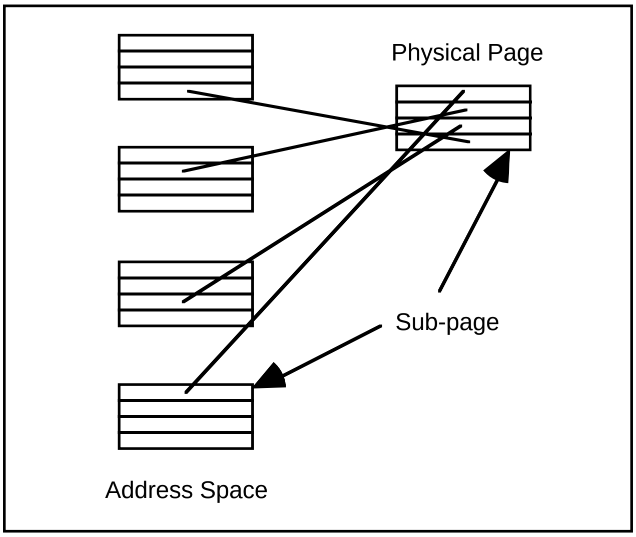
5.3: Sub-page management

A domain manager is limited by the memory management unit. Mapping physical resources are normally constrained by the page size defined by the MMU. Access permission control normally has similar constraints. Our memory management unit supports sub-page permissions. This allows us to control access permissions with a 1 KByte granularity instead of the 4 KByte granularity defined for address mapping. By convention we will define a sub-page mapping as follows:

$$\mathbf{v} \rightarrow (\mathbf{p}, \mathbf{perm} = (\mathrm{perm}_0, \mathrm{perm}_1, \mathrm{perm}_2, \mathrm{perm}_3))$$

Where **v** is the virtual address, **p** is the physical address, and the four element set **perm** specifies the sub-page permissions.

Figure 5. Sub-page management



Sub-page permissions can be used to implement a form of sub-page storage management. This allows us to allocate memory in sub-page units rather than in page sized units, leading to substantial memory savings. In our system, pages are four times the size of sub-pages. In some situations (like stack management), we see a better then four times improvement in memory utilization. Sub-page storage management is achieved by mapping a single page of storage into multiple virtual addresses and using the sub-page permissions to assure a one-to-one sub-page mapping. By one-to-one we mean that for each physical address, there is at most one virtual address mapping to it (the inverse relationship is assured by the single address space model). Figure 5 shows the effect we would like to achieve: each physical sub-page is associated with a single virtual address. Table 3 shows how permissions are manipulated to assure the one-to-one relationship required for proper sub-page storage management. The table shows four mappings to the same physical page and their associated permissions. Note that for each sub-page, there is only one mapping with a non no-access (N) permission. This mapping is the owner of the associated sub-page.

Table 3. Permission management

virtual	permissions
V_0	(N, N, N, R)
V_1	(N, RW, N, N)
V_2	(N, N, R, N)
V_3	(RW, N, N, N)

The tricky part of this approach arises when it becomes necessary to add additional sub-pages to a mapping. For example, when a stack grows it would be economical if it could be grown in sub-page increments. What do you do when the stack grows and the next sub-page in the underlying mapping is already taken by another mapping? In this case the sub-page manager needs to shuffle data between pages so that there is a physical page available with free sub-pages available where needed. The sub-page manager handles this task by maintaining an occupancy table for all pages. It computes a strategy that involves the least amount of data movement and then executes that strategy.

The sub-page management approach requires that sub-pages be used with equal probability. Otherwise, storage will not be maximally utilized. For example, the stack manager uses sub-page management to grow stacks in sub-page increments. Imagine that all stacks are a sub-page in size. If this were true, then sub-page management would only work when stacks were equally distributed on sub-page boundaries. If the stack manager were to place all stacks on a page boundary, then they would all need the zeroth sub-pages; the other sub-pages would never be used (we are assuming sub-page sized stacks). To avoid this problem, the stack manager distributes stacks over the different sub-page boundaries. This is easily done at stack creation time by adding a progressive decrement (in sub-page units) to a task's top of stack.

6: Object storage

The Newton MessagePad does not contain a permanent secondary storage device. In the base configuration, user data is stored directly in main memory (which is backed up by a lithium battery). Users can purchase a number of PCMCIA storage cards. These cards are either FLASH or battery backed-up SRAM cards. We have developed a storage model that supports these devices.

The storage model is based on objects rather than files. The storage model was designed to support medium sized objects (on the order of a dozen to a few hundred bytes). Objects can reference each other. Most importantly, the object store supports semi-transparent persistence. This means that objects can be moved in and out of permanent storage without any application code. This greatly simplifies the developers task.

Finally, the storage model is transactional. A transactional store supports committing a set of operations on the store. If an unrecoverable failure occurs during a transaction, the transaction can be aborted. This formalism assures that the store will remain consistent across a wide range of failures.

7: Loading code and data

Traditional computers deliver software on removable media such as floppy disks (and increasingly, CD-ROM). Because the Newton MessagePad lacks a removable storage device, it must support multiple distribution mediums.

This is done by defining a standard distribution format that is supported by different media types. Both stream-based and memory-based media types are supported. This allows us to distribute software on a variety of media.

Using the stream-based packages, it is possible to distribute software over the phone lines, over a computer network (such as AppleTalk™), or over a serial cable.

Using memory-based packages, it is possible to distribute software on PCMCIA ROM, RAM, FLASH, and I/O cards. In the future, disk and CD-ROM devices can also be supported.

7.1: The package model

A package consists of an index and a number of parts. The index contains an entry per part. Each entry contains information about each of the parts: a type, a type specific information field, version information, and the parts location in the package.

The part type specifies what kind of data the part represents. For example, a part could be executable code or a font.

The type specific information field specifies information about the part in a type specific format. For example, the information about a font might be an ASCII string with the font name and size. After the index comes the parts themselves.

Packages can be loaded and unloaded. When a package is loaded, the package manager is called passing it the media source. This can be either a stream-based source or a memory-based source. In either case, the package manager reads the index and processes each part entry. Assuming a part is suitable for loading (i.e., it has a reasonable version number etc.), it get dispatched to a part handler. Parts are dispatched to handlers by type. For example, a font would be dispatched to a font manager. The part handler is given the media source and told some data about the part (such as it's size, information field, etc.). It is the part handler's responsibility to process the part.

Unloading is the opposite process. The package manager informs the part handlers that a particular part needs to be unloaded. It is the part handler's responsibility to remove it from the system.

7.2: The smart serial cable

The smart serial cable protocol demonstrates a creative use of packages. We have developed a smart serial port protocol that "chirps" in an identifiable way when plugged into a Newton. The serial subsystem recognizes this chirping and begins a handshaking protocol with whatever device is connected. The protocol specifies that as part of the protocol, the device download a package. This package should contain a driver for the device and whatever other resources are necessary (such as fonts and applications). The driver is then initialized and the device comes available to the user. Our smart printer cable operates in this manner. The package is stored in ROM on a single-chip microprocessor that provides the functionality of the device.

7.3: Downloading system updates

Downloading system updates is another example of the use of packages. Updates come in the form of a set of system patches. These patches are a type of part. Online upgrades are implemented by downloading a package and installing the patch part into the system. In this manner, users can receive the most up to date system without having to go to a dealer. In this same manner it is possible to distribute other software.

8: Bibliography

[1] Richard F. Rashid. Threads of a new system. Unix Review, vol. 4(8), August 1986, pp. 37-49.

[2] ARM610 data sheet. Advanced RISC Machines Ltd., Cambridge, England, 1992.

[3] W. Smith and R. Welland. A Model for Address-Oriented Software and Hardware. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, January 1991.