QNX SOFTWARE SYSTEMS LTD.

175 TERENCE MATTHEWS CRESCENT
KANATA, ONTARIO, CANADA K2M 1W8

TEL: 613-591-0931 • 800-363-9001
FAX: 613-591-3579
INTERNET: info@qnx.com

QNX®: Microkernel Technology for Open Systems Handheld Computing

Dan Hildebrand
QNX Software Systems Ltd.
175 Terence Matthews Crescent
Kanata, Ontario K2M 1W8
Canada
(613) 591-0931

danh@qnx.com

Abstract

The resource-constrained handheld computer or PDA (Personal Digital Assistant) presents several challenges to systems developers. Working under the assumption that POSIX and UNIX-derived OS technology cannot be scaled to fit the PDA, many developers seem to be ignoring open-systems standards and opting instead for *ad hoc*, nonstandard, nonportable APIs trimmed to fit the PDA's resource constraints. This paper discusses how the QNX microkernel OS and the Photon™ microkernel GUI provide a suitably scaled environment that offers performance, extremely flexible connectivity, upwards scalability, configurability, and adherence to open-systems standards.

*This paper was presented at the Pen & Portable Computing Conference and Exposition, Boston, MA, May, 1994.
QNX is a registered trademark, and Photon (patent pending), FTL, Rundos, Ditto, and AppBuilder are trademarks, of QNX Software Systems Ltd.

Overview

Although PDAs are vastly more capable than their pocket calculator ancestors, they still impose a number of design and resource constraints on the software that animates them. For reasons of cost sensitivity, battery life, and size restrictions, the hardware resources of a PDA are limited compared to those of conventional desktop systems. As a result, the software environments that run on PDAs need to use these very limited resources as efficiently as possible. Many of the constraints imposed by PDA hardware are also representative of the constraints encountered in creating software for embedded systems.

The QNX microkernel OS, which has been very successful at bringing open-systems standards to embedded systems [Hildebrand 93], also brings these standards to PDA computing environments.

Why Open Systems Standards?

Open-systems standards are necessary in the emerging PDA environment if we are to avoid the huge costs our industry will sustain if various incompatible APIs proliferate. Without standards, applications won't be portable from one PDA to another, and the PDA marketplace will go through several generations of upheaval until market leaders and defacto standards emerge. We've already seen this painful process at work in the desktop world. While the creation of standards too early within an emerging technology can stifle innovation, PDA technology already encompasses several common, underlying services for which standards already exist (multitasking, networking, general file and device I/O, etc.). With an open-systems implementation of these underlying services, PDA vendors will be free to innovate in the areas unique to the PDA environment, and free from having to "reinvent the wheel" for technologies already well standardized. This will yield greater portability, not to mention the better variety—and quality—of applications and development tools that will result from the closer integration with desktop technologies.

Since programmers are already at home in desktop software environments, it would be unfortunate to create "yet another" programming environment for the PDA—except in cases where no suitable technology can be modified to fit. Various aspects of desktop technology need to be considered for the PDA: development environments, programming languages, operating system APIs, GUIs, and support for various I/O subsystems. Where the desktop and PDA hardware share common requirements, existing standards should be used. New technologies should be created only in areas where the requirements differ.

Although today's PDAs are severely resource constrained, they will become less so with time. As hardware resources increase, users will expect more functionality from PDAs, including support for desktop and network server-class applications. Hopefully, we won't see proprietary APIs being created for resource-constrained reasons and then promulgated with successive "API enhancements" intended to add "yet one more feature" to each iteration of the PDA systems software! Rather than extend proprietary APIs, the better approach would be to start with a subset of the open-systems environment as a core functionality, modify it as necessary for the PDA market, and then scale that environment up as resources permit. The QNX microkernel architecture is one example of how this flexibility in OS size and functionality can be achieved.

Open-systems standards are well suited to the PDA market for several reasons:

1) DOS/Windows doesn't provide sufficient functionality or "fit." With the need to support networking and background communications activities (some of which will be multimedia), a highly efficient, preemptive, realtime, multitasking OS is needed. For PDAs based on an x86-family processor, 16-bit DOS simply doesn't provide these capabilities. Environments layered on top of DOS suffer from the limitations of a DOS subsystem and require excessive amounts of memory. With the new generation of 32-bit 386-derived (or better) PDA processors, and with the performance demands imposed by handwriting and voice-processing technologies, a

© QNX Software Systems Ltd.

32-bit OS is vital to best take advantage of the hardware. The open-systems POSIX standards, as implemented in QNX, detail a very functional multitasking, multiuser, realtime, 32-bit API that can be readily scaled for PDAs.

2) Source code would be portable. Many classes of applications running on desktop systems will be useful on PDAs as well. After accounting for user-interface differences, a significant portion of the application source code would be platform neutral. For example, rather than port a database engine to a PDA-specific programming language and PDA OS API, the port would be significantly easier if the PDA's OS API conformed to the database's original API. This portability could also extend to other vendors' PDAs, allowing more and more applications to be ported. PDAs currently demonstrate poor or non-existent portability of applications; being able to run the same applications both on the PDA and on a desktop platform would be very convenient.

3) Development staff would be portable. With an open-systems API for PDAs, programmers experienced with these standards can easily apply their skills to PDA environments. Familiarity with open-systems development tools and programming languages can minimize project start-up times. Given that this is a new marketplace for which most of the needed applications have yet to be written, developers will be able to easily hire competent staff to help produce applications more rapidly. By using open-systems standards, a single development effort could readily produce both a desktop and a PDA version of an application.

4) Native development tools could be used. A pen-equipped workstation or PC running an open-systems OS can be a functional superset of the PDA. As a result, PDA applications can be developed natively on the workstation or PC, using mature and stable desktop tools without having to resort to a less-functional "cross-development" environment. By using visual and textual tools to directly create, compile, and debug PDA applications on a workstation or PC, developers can minimize their development effort and reduce the time-to-market. With a cross-development environment, developers are forced to download each experimental version of the application to the PDA for testing, resulting in a tedious cycle of "compile, download, debug" that can seriously hamper productivity. With QNX, development and runtime environments are already blended for embedded systems work [Hildebrand 93]; the same holds true for native PDA application development.

PDA Connectivity

The portable nature of the PDA creates a need for flexible connectivity. The network interface of choice is wireless, which imposes limitations on data rate and reliability. Lightweight protocols that can operate efficiently on these channels are necessary. Although irrelevant for standalone operation, the flexibility and transparency of the network interface are also important when the PDA is used as a networked extension of a desktop system or LAN. A cellular-phone link might also serve as a wide-area network for the PDA, but the bit rate for phone links is limited. While wireless network links provide 2 Mbits per second or more, the intermittent nature of the connection imposes the need for intermittent file-update protocols [Howard 93]. For very limited applications, the standalone resources of the PDA might be adequate, but for more demanding applications, the PDA's functionality must also include LAN resources.

PDA Marketplace

The PDA marketplace can be divided into "horizontal" and "vertical" segments. The horizontal market addresses general-purpose applications (e.g. appointment schedulers, address books, etc.), while the vertical market describes PDAs running custom software, often with application-specific hardware extensions. Although the general "information appliance" horizontal market is very price sensitive and limited in functionality (and therefore market penetration), the vertical market accommodates higher hardware costs, which are justified by the

© QNX Software Systems Ltd.

efficiency and time benefits that users derive. The already significant market penetration and pent-up demand for vertical-market PDAs should therefore come as no surprise.

Hardware additions for vertical-market applications include such devices as barcode scanners and transducers for measuring temperature, pressure, voltage, pH-level, light-level, heart-beat, and other physical quantities. With these hardware extensions, PDA applications face very few limits. PDAs architected as "passive-backplane PCMCIA card cages" may end up as the most flexible approach for industry-specific applications. To support this wide range of peripheral devices, flexibility in supporting and implementing custom OS extensions is needed.

Applying Microkernel Technology to the PDA

PDAs can vary widely in hardware resources—they may come equipped with a small or large memory capacity, with or without mass storage (disk drives or non-magnetic media such as flash memory), with or without a network interface, with or without a phone/modem/fax interface, with or without the capacity for handwriting or voice recognition, and with or without special interfaces for industry-specific applications. Clearly, an OS to support this platform must be very scalable. It must be able to address the PDA as a tiny, ROM-based embedded system with only the simplest of peripherals, or it must handle a PDA with many megabytes of RAM/ROM, perhaps 100 Mbytes of more of PCMCIA hard disk, high bit-rate wireless network links, cellular-phone links, and so on. The presence of a PCMCIA interface and the user's ability to exchange PCMCIA devices while the PDA is powered up require complementary capabilities in a PDA operating system. The OS should be able to mount/unmount entire OS subsystems such as filesystems and network managers "on the fly," as PCMCIA hardware is exchanged [Theimer 93]. This frees up precious internal resources for use by applications when those peripherals are either not attached to the PDA or not in use. The QNX microkernel OS provides a natural mechanism for addressing this dynamically changing range of needs.

The Microkernel Philosophy

The basic idea behind a microkernel architecture, be it an operating system or a graphical user interface, is to extract the simplest, core services from which the higher-level functionality of the environment can be constructed. Having extracted this core functionality, the resulting microkernel itself can serve as a relatively simple starting point for the "complexity" of implementing that kernel for maximum efficiency. The "art" of designing a microkernel is deciding which primitives are essential in the kernel, and which should reside outside the kernel. Although difficult to quantify, the "elegance" of the design is important. With an appropriate design, many of the functional goals the system must meet become clean extensions needing no extensive modification to the underlying architecture (i.e. kludges). One mark of a good microkernel design is the ultimate flexibility of the system as demonstrated by its ability to allow the clean implementation of extensions not anticipated in the original design.

While pursuing the theoretical ideals of a "clean" and "elegant" design, the reality of the system's performance and memory-requirements cannot be ignored. The "engineering" of the architecture requires the constant analysis of conflicting trade-offs to know where pursuing ideals adversely affects other, necessary design goals of the system (performance, flexibility, etc). A factor in assessing the elegance or cleanliness of a design is appreciating the work the hardware must perform in order to carry out the requests made of it. A good microkernel design exhibits good performance through minimizing the layering and resource contention within the design, allowing application-level requests to be serviced with minimal contortions and gyrations through the supporting microkernel architecture. An architecturally "flat" system can help deliver performance where an excessively layered design might suffer. Given the performance constraints of the PDA, these concerns are important.

© QNX Software Systems Ltd.

The QNX Microkernel Operating System

While open-systems standards have primarily evolved from existing UNIX practice, these standards don't necessarily require the existence of a UNIX kernel in order to provide appropriate services. For PDA applications, the traditional UNIX monolithic kernel architecture is far too feature-rich (i.e. large) and lacks the dynamic reconfigurability required of a PDA OS. A microkernel OS can provide these standards, not to mention a greater degree of scalability through the dynamic inclusion or exclusion of OS services. QNX adopts the approach of providing OS services via separate, optional processes. The dynamic nature of these processes allows QNX to be scaled, on demand, in order to accommodate differing peripheral and connectivity needs. In spite of its radically distinct underlying architecture, QNX is a fully POSIX-certified operating system.

The QNX microkernel is approximately 10 Kbytes in size and is comparable to a small realtime executive, both in terms of the services provided and the realtime performance of those services. The microkernel uses 14 kernel calls to implement four basic services:

- interprocess communication (IPC via message passing)
- process scheduling
- interrupt dispatching
- network message redirection

These IPC services are used to implement a lightweight RPC, and invocations of local and network-remote services are implemented in a similar manner. All system services are built upon this lightweight RPC and exhibit performance comparable or superior to their monolithic kernel equivalent [Hildebrand 92].

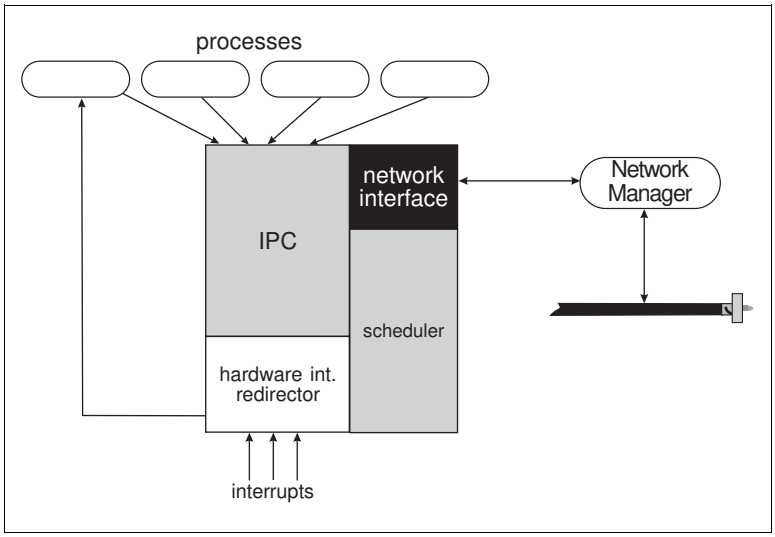

Figure 1. The QNX Microkernel.

The microkernel differs from a realtime executive in that user-level processes can provide additional services, which then appear to other processes as seamlessly integrated OS capabilities. As a result, the functionality of the operating system can be extended in any manner desired—if an SQL server is started, the OS would appear as if native SQL services formed an integral part

of the OS itself. Even highly PDA-specific information management services such as a content-addressable, object-store "filesystem" can be implemented as add-on resource managers. The microkernel OS can be scaled by launching additional processes until the full functionality of a workstation OS is achieved, including the X Window System, Motif, TCP/IP, NFS, fault-tolerant distributed networking, DOS/Windows emulation, CD-ROM support, etc. Of course, the PDA user wouldn't elect to run all these modules; the point is that the OS expresses a large range of dynamically configurable functionality.

QNX also provides a robust environment by executing the individual components of the operating system as separate, MMU-protected processes. Even as developers add experimental extensions to the OS, those extensions don't blend into the kernel in an unprotected manner; the OS retains its robust character in spite of experimental extensions. And if the experimental extensions "crash and burn," they won't bring the OS down with them. The inherent performance of QNX doesn't require the runtime "folding" of these add-on processes into the kernel address space for better performance—as is done in some microkernel systems—at the expense of overall system robustness and dynamic scalability.

QNX Resource Managers: The Process Manager

One component process of QNX is the process manager (**`Proc`**), which provides services for process creation, process accounting, memory management, process environment inheritance (for both local and network-remote processes), and pathname space management. For example, an embedded QNX system can be built from a microkernel, a process manager, and one or more application processes. The OS component of this minimal system (which includes a shared library) is 267 Kbytes, since the filesystem, device, and network I/O managers are optional. First-level pathname space management is handled by **`Proc`** because a filesystem process is purely optional under QNX.

And sufficiently privileged user-level processes can make standard API calls to adopt responsibility for portions of the pathname space and become a "resource manager." Until resource managers begin execution, **`Proc`** owns the entire pathname space, including the root and everything beneath it. As resource managers adopt their respective pathname spaces, **`Proc`** becomes responsible for maintaining a prefix tree to track which processes own various portions of the pathname space.

With **`Fsys`** (the filesystem manager), **`Dev`** (the device manager), and a few other resource managers running, the prefix tree might look like this:

`/`	Disk-based filesystem (**`Fsys`**)
`/dev`	Character device system (**`Dev`**)
`/dev/ser1`	Serial port managed by UART driver attached to **`Dev`**
`/flash`	Flash filesystem in a PCMCIA module
`/objects`	Content-addressable object-store managed by an "object database"

By implementing individual domains of responsibility within the pathname space, the components providing the OS functionality become runtime-optional services. Since resource managers live outside the kernel space and are fundamentally identical to user processes, they can be added or removed dynamically at run time, without requiring the kernel to be relinked to contain different levels of functionality. As PCMCIA modules are inserted and removed from a PDA, a manager process can automatically start and stop the processes managing that hardware, causing corresponding changes to the pathname space. Processes that control special hardware

© QNX Software Systems Ltd.

like voice I/O, barcode readers, etc. can easily adopt names in the pathname space and become visible to other processes.

The network transparency of this namespace allows the remote execution of processes to be logically equivalent to local execution. The individually administered pathname spaces blend seamlessly into the OS and across the network, behaving like native extensions of the OS itself. Full network inheritance of the entire parent-process environment (including open file descriptors, environment variables, and the current working directory) is implemented in QNX such that IPC and file I/O operate in accordance with the POSIX 1003.1 specification, ensuring network transparency.

Filesystem Managers

Since filesystem managers are just user-processes, many can be run at the same time, and each can be optimized for a specific application need. For example, a filesystem implementing a compressed, content-addressable object-store could be easily run beside, or in place of, any of the other QNX filesystems available.

The existing filesystems include **`Fsys`**, a high-performance POSIX-compliant filesystem; **`Dosfsys`** for accessing DOS media; **`Iso9660`** for CD-ROM; and the Microsoft Flash Filesystem 2 (FFS2). All of these filesystem managers are optional, and any mix can be run at the same time, or started and stopped dynamically.

Another filesystem manager is the "execute in place" filesystem manager, or **`Xipfsys`**. **`Xipfsys`** manages a chunk of ROM or Flash memory as a conventional filesystem, supporting normal filesystem operations. When the OS opens an application file for execution, **`Xipfsys`** simply maps the pages of the storage device into the OS-allocated address space and executes the application directly from ROM or Flash memory, rather than first copy the application into RAM (a limited resource). **`Xipfsys`** is also an excellent improvement over a conventional ramdisk, since the code is executed directly from the ramdisk storage area and not copied to system memory first.

Interrupt Handlers and Device Drivers

Unlike many OSs that require device-driver interrupt handlers to reside only in the kernel, QNX provides a system call that allows sufficiently privileged user-processes to connect a function to an interrupt vector. The function will then be called by the kernel, within the context of the process containing the function, in response to the specified hardware interrupt. By residing in the user process, the interrupt handler has full access to the address space of the process. Once the handler has run, it can either wake up the process it resides within or simply return to the interrupted process. QNX is a truly preemptable OS, able to reschedule processes on any interrupt event, not just on timer interrupts.

With interrupt handlers residing outside the QNX microkernel, processes acting as device drivers containing interrupt handlers can be dynamically added or removed from a running system. Interrupt handlers and device drivers aren't bound into the kernel, as in many other operating systems. In addition to dynamic flexibility, this allows device drivers to be easily debugged as user-level processes.

Network-Distributed Processing

When the microkernel needs to pass a message from a local process to a process on another node, it invokes the network manager (**`Net`**), which will use the device drivers attached to it to communicate the message to the target node. Similarly, **`Net`** can receive messages from other microkernels and route them through the local microkernel to the target process.

© QNX Software Systems Ltd.

All QNX system services are invoked via message passing through the microkernel. Since **Net** transparently connects the microkernels on the network into a single logical kernel, the result is a network of machines that behaves like a single computer. Any service available on any node can be invoked by any process, on any node. This allows the resource-limited PDA to transparently become an extension of all the computing resources of the connected LAN and WAN environments.

Like **Fsys** and **Dev**, **Net** can also be started and stopped from the command line. It also has a family of drivers and can manage multiple network drivers at once. If **Net** discovers that more than one of the network drivers provides connectivity to the same destination, it will load-balance the traffic between the connections. Multiple network paths between nodes can provide both better throughput and network fault-tolerance. Application-level changes aren't needed to take advantage of this fault-tolerance, since the support exists within the network manager itself. This functionality can benefit a PDA in a wireless LAN environment suffering from intermittent transmission problems by automatically making use of multiple connections in order to overcome communications difficulties. If a PDA user connected through a wireless Ethernet travels outside the range of the LAN, **Net** can automatically establish a cellular-phone modem connection to maintain connectivity without the application's intervention.

Net implements FTL,™ a lightweight LAN protocol intended to deliver the full performance of the LAN interface to applications. Since the generalities of heterogeneous LAN protocols like TCP/IP aren't needed for QNX-to-QNX communications, this protocol provides absolute minimal overhead for QNX network transactions. Efficiency at this level is important, since the intent of the QNX LAN is to transform the QNX nodes on the LAN into a single, logical QNX machine, using as little overhead as possible, especially with the limited bandwidth a wireless link might provide.

For communication between a QNX and non-QNX node, **Net** makes available a raw packet interface that other processes can use for hosting "guest" protocols such as TCP/IP and DOS LAN protocols. To a non-QNX network, the cluster of QNX nodes on the LAN appears to be a single, multiprocessor machine. Since the lightweight QNX FTL protocol uses standard packet headers, it peacefully coexists with other protocols on the same cable. By hosting TCP/IP, SL/IP, CDPD, and other heterogeneous protocols, QNX is able to network as a peer with non-QNX systems, but is still able to use the FTL protocol when possible for better performance and network transparency. This simultaneous use of both protocols can be very effective, as demonstrated by the X Window System under QNX, which can use the fast FTL protocol for client/server transactions between QNX nodes, and simultaneously use TCP/IP to remote clients running on non-QNX hosts [Furr 94]. As suggested elsewhere [Loucks 93], another use for the FTL protocol is to allow a PDA that isn't capable of self-hosting a resource-expensive protocol like TCP/IP to use FTL in order to gateway through the TCP/IP protocol service running on another, more resource-rich, machine.

By starting with a minimal QNX system (microkernel and **Proc**) and adding the **Net** process and a network driver, a PDA can become part of the "logical computer" formed by the other nodes on the network. Any process on the PDA can use the resources of any other node on the network as if it were a local resource. As a result, the PDA will inherit the database, processor, and other resources of the entire network; and the rest of the network will be able to communicate with the processes on the PDA. With permission, the PDA will be a true network peer able to move processing "agents" out into the LAN or accept processes from the LAN into the PDA to implement interface processes for LAN-local services as suggested in [Theimer 93]. This also allows applications that may be too resource intensive for a specific PDA to run on another node on the network and to display their output on the "local" PDA (without application changes).

© QNX Software Systems Ltd.

DOS Compatibility

For larger, resource-rich portable machines, DOS support may also be useful. Although microkernel operating systems are architecturally very different from DOS, this doesn't mean they can't support DOS applications. Given that QNX runs on PC hardware and that DOS applications are abundant, support for DOS programs can be very useful. QNX provides a Rundos™ utility that allows most DOS applications, including MS-Windows, to run within QNX. DOS applications running under Rundos use the native QNX network-distributed filesystem, which provides DOS-compatible network file locking so that DOS network applications share files appropriately. A facility for mounting DOS-native filesystems into QNX also allows both QNX and DOS applications to read and write DOS media, on any network node. DOS LAN applications also benefit from the fault-tolerant nature of the redundant networking provided by **`Net`**.

Photon™: A Microkernel GUI

Although an API suitable for the PDA's OS is well defined by the POSIX standards, a GUI standard appropriate to the PDA doesn't exist, MS-Windows and the X Window System notwithstanding. Neither MS-Windows nor X can be scaled small enough to fit within the resource constraints of a low-end PDA. Given the success of the QNX microkernel approach to scale the open-systems OS API to the PDA, we have used a similar approach to create Photon, a scalable GUI for the PDA. With an interface library for X that supports the same binary API as the Photon interface library, applications that run on the PDA under Photon can also be used on any X-equipped workstation or PC, thereby qualifying Photon as an "open-systems GUI." With an interface library for Photon that borrows from the widget-library concept of X (but scaled down for the PDA), we have an API that is robust, extensible, and already familiar to open-systems programmers.

To successfully implement a high-performance microkernel OS, we first had to tune the microkernel so that the kernel calls for IPC were as lean and efficient as possible, since the performance of the whole OS rested on this capability. Having implemented this low-overhead IPC, we were able to structure a GUI as a graphical "microkernel" process surrounded by a team of cooperating processes communicating via that low-overhead IPC. While this might seem at first glance like building a graphical system around the classic client/server paradigm already used by X, the approach differentiates itself in the functionality implemented within the graphical microkernel (or server) itself.

Running under QNX, the Photon microkernel GUI implements only a few fundamental primitives from which the higher-level functionality of a windowing system is constructed by external, optional processes. Ironically, for the Photon microkernel itself, "windows" do not exist. Nor does the Photon microkernel possess the ability to draw anything, or manage a pen, mouse, or keyboard. Instead, Photon creates an "event space" and confines itself only to managing "regions" and processing the clipping and steering of various events as they flow through the regions in this event space. This abstraction is roughly parallel to the concept of a microkernel OS, which by itself is incapable of filesystem or device I/O and must rely on external processes to provide such high-level services. Just as a microkernel OS can be scaled up or down in size and functionality, so also can a microkernel GUI.

The core "abstraction" implemented by the Photon microkernel is that of a graphical event space that other processes can populate with regions. These other processes use the native QNX IPC to communicate with the Photon microkernel, and manipulate their regions to provide higher-level graphical services or to act as user applications. Removing service-providing processes allows Photon to be scaled down for limited-resource systems; adding service-providing processes allows Photon to be scaled up to full desktop functionality. The underlying efficiency of the QNX IPC and the architecture of Photon can meet the efficiency and performance requirements

© QNX Software Systems Ltd.

to run well on low-end PDA hardware, while also maintaining the scalability and flexibility needed for larger systems.

The Event Space

The event space managed by the Photon microkernel can be visualized as an empty void with a device plane up front that the user "looks into." At the very back of this event space is a root plane. Applications place regions into the three-dimensional space between these two planes and use those regions to generate and accept various types of events within this space. Processes that provide device-driver services place regions in front of the device plane. In addition to managing the device and root plane, the Photon microkernel maintains an onscreen pointer projected as draw events from the device plane towards the user.

Events (which can be looked upon as the "photons" from which this windowing system gets its name) travel through this event space. Events themselves consist of a list of rectangles with some attached data. As the events flow through the event space, their rectangle lists intersect the regions placed there by various processes (applications). Events travelling away from the root region of the event space are said to be travelling out towards the device plane and the user, while events issuing from the user are said to be travelling in towards the root plane at the back of the event space.

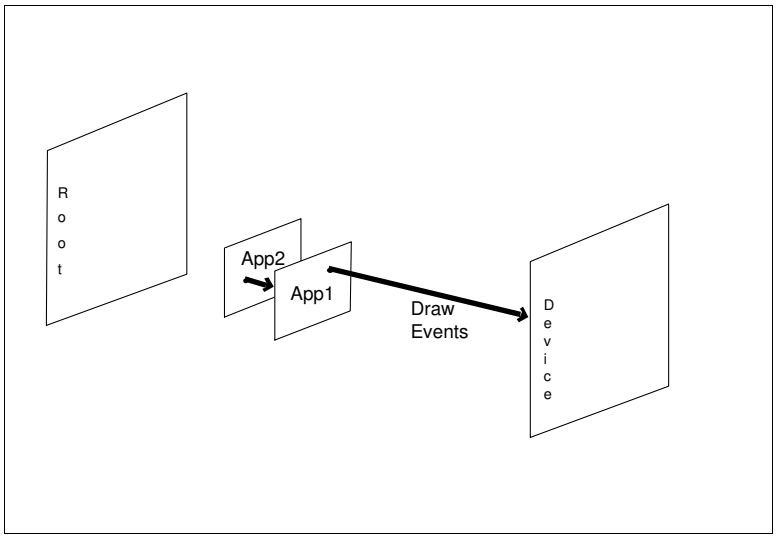

Figure 2. Photon event space with root, application, and driver regions.

The interaction between events and regions is the basis for the input and output facilities in Photon. Pen, mouse, and keyboard events travel away from the user towards the root plane, with the location of their associated onscreen pointer. Draw events originate from regions and travel towards the device plane and the user.

Regions

Regions are managed in a hierarchy and associated as a family of rectangles that define where they exist in the three-dimensional event space. A region also has attributes that define how it

© QNX Software Systems Ltd.

interacts with various classes of events as they intersect the region. The interactions a region can have with events are defined by two bitmasks: a *sensitivity* bitmask and an *opaque* bitmask.

The sensitivity bitmask defines which intersections by specific event types that the process owning the region wishes to be informed of. A bit in the sensitivity bitmask defines whether or not the region is sensitive to each event type. When an event intersects a region for which the bit is set, a copy of that event is enqueued to the application process that owns the region, notifying the application of events travelling through the region. This notification does not modify the event in any way.

The opaque bitmask is used to define which events the region is opaque (in the optical sense) to. For each event type, a bit in the opaque mask defines whether or not the region is opaque or transparent to that event. The optical property of "opaqueness" is implemented by modifying the event itself. These two bitmasks can be combined to accomplish a variety of effects in the event space. Four combinations are possible:

- **Not sensitive, transparent**—The event passes through the region, unmodified, without the region owner being notified. The process owning the region simply isn't interested in the event.
- **Not sensitive, opaque**—The event is clipped by the region as it passes through; the region owner isn't notified. For example, most applications would use this attribute combination for draw-event clipping, so that an application's window wouldn't be overwritten by draw events coming from underlying windows.
- **Sensitive, transparent**—A copy of the event will be sent to the region owner, and the event will continue, unmodified, through the event space. A process wishing to log the flow of events through the event space could use this combination.
- **Sensitive, opaque**—A copy of the event will be sent to the region owner, and the event will also be clipped by the region as it passes through the event space. By setting this bitmask combination, an application can act as an event filter or translator. For every event received, the application can process and regenerate it, arbitrarily transformed in some manner, possibly travelling in a new direction, and perhaps sourced from a new coordinate in the event space. For example, a region could absorb pen events, perform handwriting recognition on those events, and then generate the equivalent keystroke events.

Events

Like regions, events also come in various classes and have various attributes. An event is defined by an originating region, a type, a direction, an attached list of rectangles, and optionally, some event-specific data. Unlike most windowing systems, Photon classifies both input (pen, mouse, keyboard, etc.) and output (drawing requests) as events.

Events can be generated either from the regions that processes have placed in the event space or by the Photon microkernel itself. Event types are defined for: keystrokes, pen and mouse button actions, pen and mouse motion, region boundary crossings, expose and covered events, draw events, and drag events. Application processes can either poll for these events, block and wait for them to occur, or be asynchronously notified of a pending event. The rectangle list attached to the event can describe one or more rectangular regions. Or it can be a "point-source," that is, a single rectangle where the upper-left corner is the same as the lower-right corner.

When an event intersects a region that is opaque to the event, that region's rectangle is "clipped out" of the event's list of rectangles such that the event's rectangle list describes only the portion of the event that would ultimately be visible. The best way to see how this clipping is performed is to examine the changes in the rectangle list of a draw event as it passes through various

intersecting regions. When the draw event is first generated, the rectangle list consists of only a single, simple rectangle describing the region from which the event originated.

If the event goes through a region that clips the bottom-left corner out of the draw event, the rectangle list is modified to contain only the two rectangles that would define the area remaining to be drawn.

In a similar manner, every time the draw event intersects a region opaque to draw events, the rectangle list will be modified to represent what will remain visible of the draw event after the opaque region has been "clipped out." Ultimately, when the draw event arrives at a graphics driver ready to be drawn, the rectangle list will precisely define only the portion of the draw event that is visible.

If the event is entirely clipped by the intersection of a region, the draw event will cease to exist. This mechanism of "opaque" windows modifying the rectangle list of a draw event explains how draw events from an underlying region (and its attached process) are properly clipped for display as they travel towards the user at the device plane.

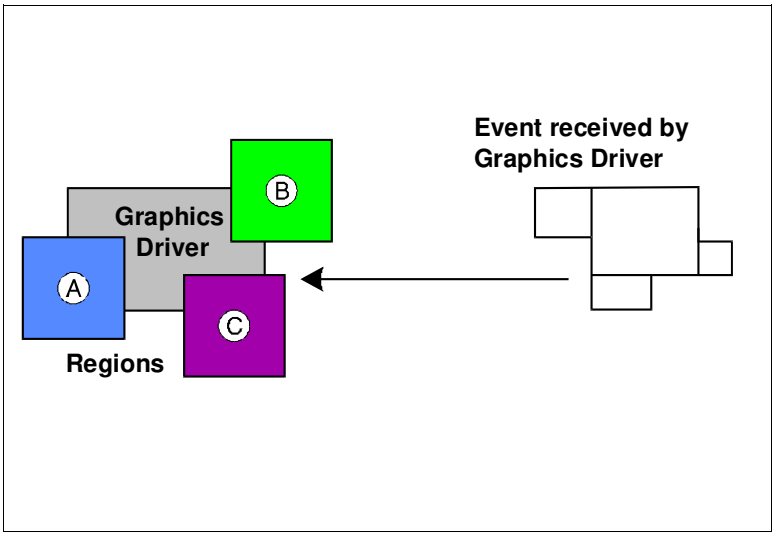

Figure 3. A draw event being clipped.

Photon Device Drivers

Device drivers under Photon are nothing more than applications that are receiving and emitting the classes of events needed to provide a "device-driver" service. Under Photon this means emitting keyboard, pen, and mouse events, and receiving and processing draw events. This point is worth repeating: Photon device drivers aren't connected in some privileged way to the Photon microkernel, but are instead simply regular application processes that generate or respond to events of specific types in the event space. As with a microkernel OS, this allows device drivers to be easily started and stopped at run time, and to be developed with the same tools (and ease of development) used for application programs.

© QNX Software Systems Ltd.

Pen and Mouse Driver

The pen/mouse driver is implemented as a process that places a region in front of the device plane. As the process gets data from the hardware, it builds Photon "mouse events" and fires them into the event space, travelling towards the root plane. As these events pass through the device plane, Photon takes care of "bouncing" a visual pointer draw event back out to a graphics driver region. In effect, the device plane is acting as a "half-silvered" mirror that allows the pen or mouse event to continue into the event space, but also bounces back a draw event for the input event so that the onscreen pointer can track the input device. The actual pointer displayed on the screen is distinct from the pen or mouse driver and can exist without a pen or mouse driver present.

No assumption is made by Photon as to what a pointing device is. The process injecting pen or mouse events can interface to any arbitrary hardware or process and can transform the collected data into input events.

Keyboard Driver

As with the pen/mouse driver, the keyboard driver is implemented as a process that places a region in front of the device plane. As it gets keystrokes from the hardware, it builds Photon "keyboard events" and fires them into the event space, travelling towards the root plane. As with the pen/mouse driver, Photon makes no assumptions as to what a keyboard is. The process injecting keyboard events could acquire data from any arbitrary hardware or process and inject that data as keyboard events.

Many PDA applications need to "pop-up" a keyboard that the user can operate by "tapping" the displayed keys. If this keypad is created as a region that can accept pen events and transform them into key events "injected" into the event space, any application could use this keypad without explicit programming effort. Similarly, an application that reads analog data representing speech from a **`/dev/audio`** resource manager could perform voice recognition on the data and inject the equivalent keystrokes into the system, all without application modifications.

Graphics Drivers

Graphics drivers are implemented as processes that place a region in front of the device plane, but rather than inject pen, mouse, or keyboard events, a graphics driver's region is sensitive to draw events coming out of the event space. As draw events intersect the region, those events are received by the graphics driver process. In effect, the region can be imagined to be coated in "phosphor" illuminated by the impact of "photons."

Since Photon's drawing API accumulates draw requests into batches emitted as single draw events, each draw event received by the driver contains a list of individual graphical primitives to be rendered. By the time the draw event intersects the graphics driver region, its rectangle list will represent a "clip list" that describes exactly which portions of the draw list are to be actually rendered to the display. The job of the graphics driver is to transform this clipped draw list into a visual representation on whatever graphics hardware the driver is controlling.

One advantage of delivering a "clip list" within the event passed to the driver is that each draw request then represents a significant "batch" of work. As graphics hardware advances, more and more of this batch of work can be pushed directly into the graphics hardware. Many display controller chips already handle a single clip rectangle; graphics hardware that can handle multiple clip rectangles is imminent.

Although it may appear to be an unacceptable overhead to use the QNX IPC services to pass draw requests from the application to the graphics driver, our performance measurements demonstrate that this implementation performs as well as having the application make direct calls into a graphics driver. One reason is that multiple draw calls are batched with the event mechanism, allowing the already minimal overhead of the QNX lightweight IPC to be amortized over many draw calls.

© QNX Software Systems Ltd.

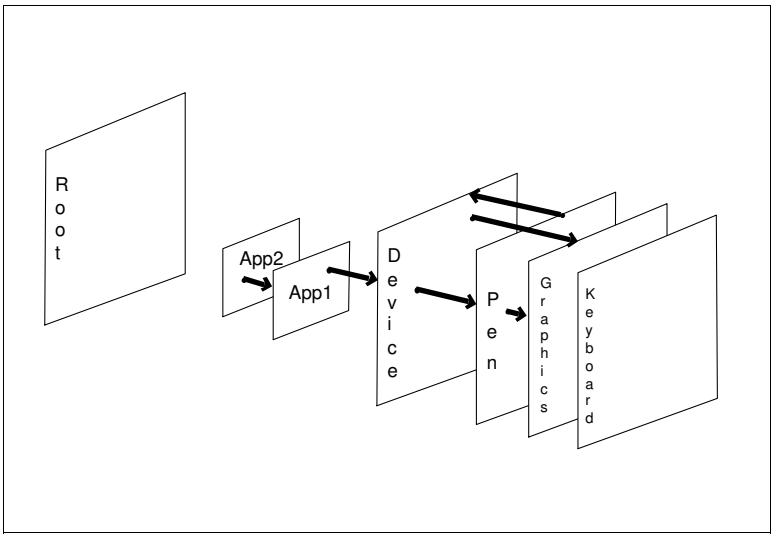

Figure 4. Photon device drivers.

Multiple Graphics Drivers

Since graphics drivers simply put a region into the Photon event space—and that region describes an X-by-Y space to be intersected by draw events—it naturally follows that multiple graphics drivers can be started for multiple graphics controller cards, all with their draw-sensitive regions present in the same event space. These multiple regions could be placed physically adjacent to each other, describing an array of "drawable" tiles, or overlapped in various ways. Since Photon inherits QNX's network transparency, Photon applications or drivers can run on any node, allowing additional graphics drivers to extend the graphical space of Photon to the physical displays of many networked computers. By having the graphics driver regions overlap, the draw events can be replicated onto multiple display screens.

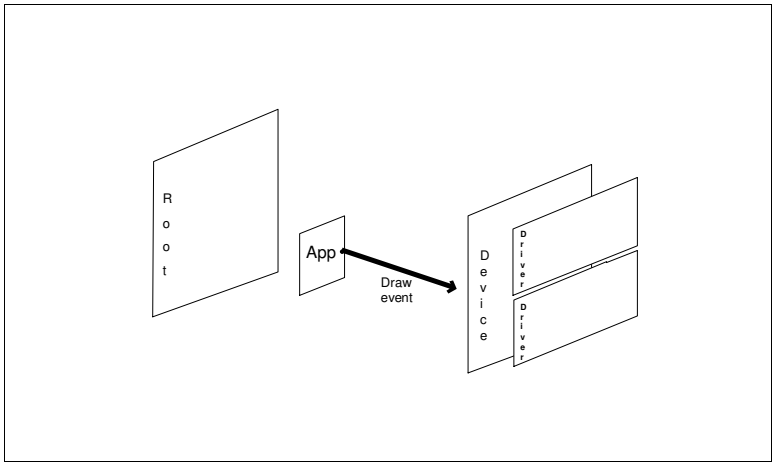

Figure 5. Multiple graphics drivers.

© QNX Software Systems Ltd.

Many interesting applications become possible with this capability. For example, a factory operator with a wireless-LAN PDA in hand could walk up to a desktop computer and drag a window from a plant control screen onto the PDA, and then walk out onto the plantfloor and interact with the control system to monitor or adjust plantfloor equipment.

Useful collaborative modes of work for people using multiple PDAs are also possible. For example, a group of people could simultaneously see the same application screen on their PDAs and cooperatively operate the application.

From the application's perspective, a whole group of PDAs and networked computers would look like a single, unified graphical space. From the user's perspective, windows can be dragged from physical screen to physical screen, even across network links.

Color Model

Colors processed by the graphics drivers are defined by a 24-bit RGB quantity, 8 bits for each of red, green, and blue, yielding a total range of 16,777,216 colors. Depending on the actual display hardware managed by the driver, the driver will either invoke the 24-bit color directly from the underlying hardware, or use various dithering techniques to create the requested color from less-capable hardware. Since the graphics drivers use a hardware-independent color representation, applications can be displayed without modifications on hardware possessing varied color models. This allows applications to be "dragged" from screen to screen without concern for what the underlying display hardware's color model might be.

Encapsulation

Since graphics drivers for Photon are just applications that process draw events, a Photon graphics driver can be implemented as a graphical application inside another windowing system. The graphics driver could also take the keyboard, pen, and mouse events it receives from the other windowing system and regenerate them within Photon, allowing the Photon "window" within the other windowing system to be fully functional, both for graphical display and keyboard/pen/mouse input. The implementation of this facility is natural and clean, requiring none of the systems-level kludge work often associated with this sort of service in other windowing systems.

Ditto™ —Remote Screen Viewing

QNX provides a utility that allows a remote desktop screen to be viewed and manipulated as if it were local. We call this facility "Ditto." This is useful for remote diagnostics, technical support/training, collaborative team work, and many other situations.

A problem with trying to implement a Ditto application for *graphical* environments is that graphical screens contain potentially megabytes of pixel data, requiring a large bandwidth to relay the screen image, as well as large amounts of processor and memory overhead to compare previous screen images with current screen images in an attempt to send only differences so as to minimize the bandwidth requirement. Typically, a PDA lacks the processor, memory, and communications bandwidth to support a pixel-copying Ditto effectively. Under Photon, this is easily done.

The Photon Ditto is implemented as a transparent, draw-event-sensitive region placed in front of the entire screen (or a single application region). As draw events come out of the event space and transparently pass through the Ditto region, they would be received by the Ditto process. This Ditto process would also own a region in another Photon event space, on another node in the network. In that event space, the draw event would be regenerated, travelling out towards a graphics driver where the same draw list as on the first system would be processed and drawn.

© QNX Software Systems Ltd.

In a similar manner, keyboard, pen, and mouse events entering the Ditto region in the second Photon event space could be relayed across and regenerated in the Photon event space being monitored. The advantage of this approach is that draw events require a much lower bandwidth than pixel copying and comparing. The Photon Ditto is functional across low-bandwidth links even though its core functionality can be expressed in less than 160 lines of C source or less than 1 Kbyte of code.

Printer Driver

To print all or part of a Photon screen, a print application need only put a region (opaque but sensitive to draw events) in front of either the entire screen or a single application's region. An expose event would be sent towards the underlying application(s), and the resulting draw events would be received. Once the draw events are captured, the print region can be removed without causing a visible refresh on the screen. If an application region buried deep in the event space needs to be printed, the print region could be buried equally deep in the event space to cause an expose only on that region. With the expose-triggered draw events captured, the print application can translate them into the format necessary for the print device. If the print device is a PostScript printer (connected to another computer accessible via the LAN), the draw events can be translated directly into PostScript and printed. Since the draw events being translated are high-level draw requests, they can be rendered on the print device at full printer resolution, rather than with the coarse pixelation that results from a screen-resolution pixel dump.

The Photon Window Manager

As an example of the scalability inherent with a microkernel GUI, adding a window manager to Photon creates a fully functional, desktop-class GUI. The window manager is entirely optional, and for most classes of PDA applications, completely unnecessary. If present, the window manager allows the user to manipulate application windows by resizing, moving, and iconifying them. The window manager is built on the concept of filtering events with additional regions placed behind application regions upon which a title bar, resize handles, and other "gadgets" are drawn and interacted with. Since the replaceable window manager implements the actual "look and feel" of the environment, various user-interface flavors can be configured or installed.

Memory Requirements

For a 32-bit QNX microkernel, process manager, and the supporting shared libraries, the memory requirements total 267 Kbytes. With this shared library in place, additional QNX processes save significantly on their size, being able to use code from the system shared library.

	Code	**Data**	
QNX Microkernel	10K	0K	
Process Manager	100K	100K	
QNX Shared Library	57K	0K	
	167K	100K	= 267 Kbytes for the QNX OS

By adding Photon, its shared library, and some device drivers, an additional 248 Kbytes are required:

© QNX Software Systems Ltd.

	Code	Data	
Photon Microkernel	20K	40K	
Photon Shared Library	86K	0K	
VGA Graphics Driver	50K	20K	
Pen Input Driver	16K	16K	
	172K	76K	= 248 Kbytes for Photon

Since the window manager is purely optional, and likely unneeded on a PDA, it doesn't need to be included in the default Photon memory requirements.

	Code	Data	
Photon Window Manager	30K	64K	= 94 Kbytes for the Window Manager

The PDA has the option of accessing a local filesystem device, of networking to a remote filesystem, or of providing multiple combinations of these.

	Code	Data	
Network Manager	24K	24K	
Ethernet Driver	20K	20K	
	44K	44K	= 88 KBytes for networking

In the case of filesystems, the wide variety makes it difficult to characterize the memory requirements. In general, however, a filesystem is implemented in less memory than the network manager and network driver.

	Code	Data	
PCMCIA Flash filesystem	20K	16K	= 36 Kbytes for a local filesystem

With a ROM-based PDA, much of the code for QNX and Photon can be moved to ROM, leaving a relatively small component of QNX and Photon in RAM. The remaining RAM in the PDA can be used to run transient applications, loaded either from a local filesystem or through the network connection. Additional applications can also be run directly from the PDA's ROM, using the RAM as a workspace, or remotely in the memory of another node on the network.

Development Environment

As a platform for software development, most PDAs are insufficient to host the caliber of tools the modern software developer is accustomed to. This necessitates the use of a desktop machine equipped with the appropriate tools to facilitate PDA application development. For development, the QNX/Photon environment allows PDA applications to run on the desktop machine under either Photon or X, so that debugging and usage-testing can be done without the PDA present. Once the application has evolved to the point that testing on the PDA itself is required, *any* network connection to the PDA will allow the developer to "launch" the application onto the PDA, while the full-screen symbolic debugger and other tools running on the desktop system allow the developer to continue to debug the program exactly as if it were still running on the desktop. A useful by-product of this development environment naturally presents itself here: a

single development effort can produce an application targeted at both the PDA and the desktop system.

Programming Languages

Some PDAs have implemented interpreted languages for application development. Reasons for this choice are that interpreted languages can provide good code density compared to compiled languages and that the underlying processor can be changed without requiring recompilation of application programs. However, interpreted languages derive these benefits by invoking subroutines within the interpreter for the various "tokens" recognized in the program text. This imposes the penalty of having computationally intensive code slowed down by the interpreter.

Another approach is to use a conventional compiled language like C or C++, but implement the runtime environment with a rich shared library. The shared-library routines will end up being roughly equivalent to the interpreter's runtime subroutines, yet the application program itself would be optimized machine code. The resulting compiled applications would be small, but would also retain the compute-intensive performance of the underlying PDA CPU. The maturity of the development tools and richness of the associated tools for compiled languages would also favor this approach. Although it remains to be seen whether or not PDA applications will be compute-intensive enough for this difference to matter, retaining this performance advantage leaves future capabilities unfettered.

Another real advantage of using compiled languages is that significant extensions to the OS such as data transfer protocols and device drivers can be routinely coded with the standard PDA development environment. It would be hard to imagine accomplishing this class of extension with an interpreted language. If special, machine-level tools were made available for primarily interpreted PDAs, the dichotomy of the two classes of tools would raise support and maintenance issues as well. Without machine-level access to development tools, a PDA is a "closed" system bearing little or no resemblance to an "open-systems" participant.

Application Builder

Setting aside the issue of programming languages, the large number of distinct, vertical markets will require the development of many unique, customer-specific applications. The graphical nature of these applications, and the human-factors issues related to their ease-of-use, will mandate employing graphical development environments that enable "rapid prototyping." For the effective prototyping of GUI applications, a visual development environment is required, together with the ability to directly express the algorithms behind the visual interface in a suitable programming language. The AppBuilder™ provides this capability for QNX. And since Photon applications also run on X, the AppBuilder is capable of generating applications for X as well.

The AppBuilder provides a visual environment where the developer can "drag and drop" widgets from a palette containing both Photon-standard widgets and user-developed widgets into the application under construction. For the events those widgets encounter, the developer can attach either additional widgets (menus, dialogs, etc.) or C-source fragments to directly express the actions to be taken. User-developed widgets can either be variations of existing Photon widgets or higher-level widgets constructed from many simpler widgets. Over time, the developer will end up with an extensive palette of previously defined widgets that are unique to his application domain (e.g. special entry forms).

Once the visual prototype of the application is built, the AppBuilder will then generate a complete, free-standing C-source version of the application and compile it, ready for execution and testing. Iterative development is supported—the developer can alternate between expanding the C-source code "underneath" the visual application and enhancing the visual appearance itself, without loss of work.

© QNX Software Systems Ltd.

A standard capability of AppBuilder-generated applications is that they can use an external file for onscreen text replacement for foreign language support. The Photon widget library itself is configurable for various look-and-feel requirements, including a 3D Motif style. The shared library containing the widgets can also be replaced with specialty versions for unusual requirements, without the need for corresponding application changes.

Performance Summary

Within the QNX microkernel, sufficient architectural optimization has been accomplished to achieve a 26 µsec context switch on a 33MHz 386 (12 µsec on a 33MHz 486). For virtually all system calls, the processing done to satisfy the system call (network I/O, etc.) outweighs the few microseconds consumed by the context switch. Having achieved this level of performance, the additional architectural overhead imposed by splitting the OS into a team of message-passing, cooperating processes becomes negligible. However, the resulting gains of greater parallelism and concurrency within the OS result in a better performing operating system in general [Hildebrand 92].

On the GUI side, applications running in the Photon event space achieve drawing speeds that are very close to the full capacity of the graphics driver and display hardware.

Future Directions

Under QNX, a process called **`Csc`**, the Client Side Cache, already exists to provide locally cached filesystem information for networked-connected machines. Given the intermittent nature of a wireless network connection, extending **`Csc`** to hold pending-filesystem transactions for later reconciliation with the cached data on other filesystems is under investigation. API extensions to allow applications to be "cache aware" during disconnected intervals of operation are also under investigation [Goldberg 93].

Databases such as SQL servers already exist for QNX; these are able to maintain a local transaction log for transactions against a locally cached subset-version of a central database. When network connections are "up," the transaction logs flow through to the central database; but while the network connection is "down," the user of the PDA with the local database continues to work as before. Compression management within the PDA-cached database and transaction log minimizes the resources required in the PDA.

Conclusion

The QNX microkernel OS provides the scalability needed to effectively bring the highly functional open-systems standards to resource-constrained PDA hardware in a high-performance manner. Similarly, the Photon microkernel GUI efficiently delivers the scalability needed to address GUI applications for tiny ROM-based PDAs up through X-equipped workstations. Borrowing from open-systems standards, QNX's rich and familiar development environment enables the rapid development of the wide range of applications that will be needed to address the already growing application needs of PDA users. For the future, Photon's flexibility coupled with QNX's network-distributed transparency presents interesting new capabilities for applications to build on—the entire resource set of a LAN can be delivered to the PDA in the palm of the user's hand.

© QNX Software Systems Ltd.

References

[Furr 94] Steve Furr. *Efficient X Server Connections in a Realtime Message-Passing Environment*. Xhibition '94 Conference Proceedings, San Jose, CA (to be published June, 1994)

[Goldberg 93] David Goldberg and Michale Tso. *How to Program Networked Portable Computers*. IEEE Fourth Workshop on Workstation Operating Systems, Napa, CA, October, 1993, pp. 30-33

[Hildebrand 92] Dan Hildebrand. *An Architectural Overview of QNX*. Proceedings of the Usenix Workshop on Micro-Kernels & Other Kernel Architectures, Seattle, WA, April, 1992, pp. 113-126

[Hildebrand 93] Dan Hildebrand. *A Microkernel POSIX OS for Realtime Embedded Systems*. Proceedings of the Embedded Computer Conference and Exposition, Santa Clara, CA, April, 1993, p. 1601

[Howard 93] John H. Howard. *Using Reconciliation to share files between occasionally connected computers*. IEEE Fourth Workshop on Workstation Operating Systems, Napa, CA, October, 1993, pp. 56-60

[Loucks 93] Larry Loucks, Ravi Manikundalam, and Freeman L. Rawson III. *A Microkernel-Based Operating System for Personal Digital Assistants*. IEEE Fourth Workshop on Workstation Operating Systems, Napa, CA, October, 1993, pp. 9-13

[Theimer 93] Marvin Theimer, Alan Demers, and Brent Welch. *Operating System Issues for PDAs*. IEEE Fourth Workshop on Workstation Operating Systems, Napa, CA, October, 1993, pp. 2-8

© QNX Software Systems Ltd.