By Curtis A. Schwaderer,
Director of Network Technologies,
Microware Systems Corporation.

Network Processors and their Impact on Real-time Operating Systems

Network usage is growing exponentially, pushing existing communications infrastructure to their performance and capacity limits. Communications product development cycles are beginning to take longer than the service life of the equipment being developed. Network processor technology promises shortened development cycles, greater flexibility, and longer product service life. The truly amazing part of this revolution is that everyone seems to be overlooking the impact network processor technology has on today's real-time operating systems. In the sections that follow, we'll look at network processor technology and new requirements network processors place on a real-time operating system.

INTRODUCTION

We stand at unique historical crossroads in the communications industry. Network topologies are pushing data rates to gigabit/sec levels. Networking protocols designed to accommodate voice, data, or multimedia traffic emerge almost daily. Network usage is growing exponentially, pushing existing communications infrastructure to their performance and capacity limits. Communications product development cycles are beginning to take longer than the service life of the equipment being developed.

Network processor technology promises shortened development cycles, greater flexibility, and longer product service life. The truly amazing part of this revolution is that everyone seems to be overlooking the impact network processor technology has on today's real-time operating systems. In the sections that follow, we'll look at network processor technology and new requirements network processors place on a real-time operating system.

OVERVIEW OF NETWORK PROCESSOR TECHNOLOGY

The network infrastructure has become a series of fixed-feature products, ultimately creating a "fixed service" network. The resulting network is fundamentally limited in the kinds of services that can be deployed without expensive overhaul of the network. This challenge has caused network operators and communications device manufacturers to look for a more programmable approach while still achieving higher levels of wire speed performance.

Network processors eliminate this inflexibility by integrating a core processor with one or many microengines. These microengines are designed with specialized instruction sets that enable very fast packet processing. The result is an "all-in-one" package that has the ability to provide wire speed performance with a flexible, programmable feature set.

Figures 1-a and 1-b contrast the product lifetime of ASIC and network processor based communications products. New advancements in communications technology may cause a hardware redesign for ASIC based designs, perhaps even before the initial product is in production. Network processors enable the development of a core network processor based hardware platform. New features are accommodated by upgrading microcode without requiring a hardware change. In addition, if the design allows for connecting companion cards into the system to support various network topologies, the same base product can be used for

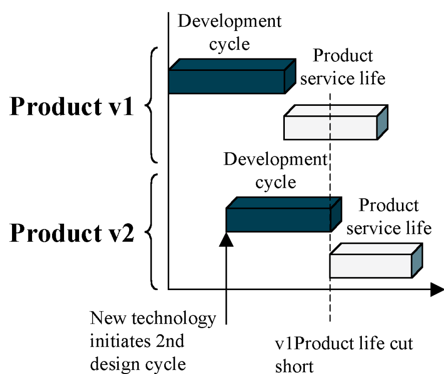

Figure 1a. ASIC based product life cycle.

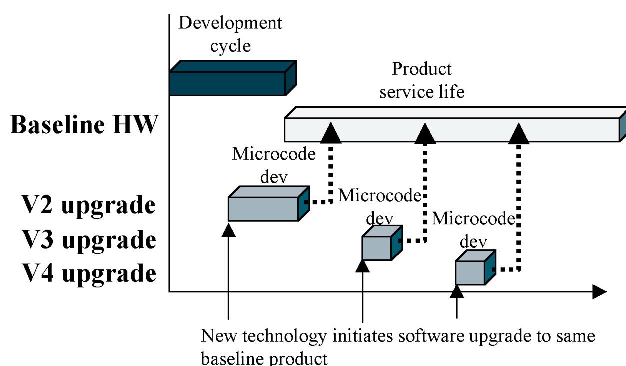

Figure 1b. Nework processor product life cycle.

Copyright 2001 by Dedicated Systems Magazine - 2001 Q2 (http://www.dedicated-systems.com)

many different product lines.

THE INTEL IXP1200 NETWORK PROCESSOR

In order to understand a little better about network processor technology, let's look at the Intel IXP1200. The IXP1200 integrates a 32 bit StrongARM core with six 32-bit RISC microengines. The StrongARM core runs at 162 MHz and is responsible for general system control and microcode download to the microengines. The IXP1200 running at 162 MHz is rated for 2.4 million "mini-packets" (i.e. 64 byte IP packets) of sustained throughput. This translates into over 1.2 Gbps of full duplex aggregate throughput.

The six microengines run independently with local register sets to run efficiently. Each microengine runs code out of their own 1Kword control store. Each microengine can support up to 4 threads for a total of 24 packet processing threads per IXP1200. The engines have zero-overhead context switching between each of their four threads and as a result, can switch threads while waiting for data.

The PHY device block comes in the form of the IX bus. The IX Bus is a FIFO-oriented, scheduled bus. The bus supports 64 bit bursts of up to eight cycles in length. The IX bus can also be configured as one 32-bit bus in each direction.

The IXP1200 supports two non-multiplexed buses in the memory interface block. The first bus is an SRAM fast memory interface. This interface is used for memory areas holding routing and QoS tables (for example). The second interface is an SDRAM that provides lower cost (and performance) mass storage area.

This kind of integrated architecture places significant functionality requirements on the software baseline. First of all, the system must provide a complete development environment and RTOS for the StrongARM core. The RTOS and software baseline must also be able to set up independent colored memory areas for SRAM and SDRAM access. The PCI bus interface must also be supported. A communications software framework must be provided that runs on the StrongARM core, but has the ability to be extended down into the microengine environment. The OS-9 RTOS supports these capabilities. In addition specific 10/100/1000BaseT and ATM solution stacks have been created that eliminate interworking code development.

THE "NETWORK PROCESSOR OPERATING SYSTEM"

Ironically, as processor and network technologies have advanced, core principals of real-time operating system design have not. The majority of embedded software providers have focused on developing toolsets for general purpose processors, not on advancing the state of the operating system foundation to better accommodate these new paradigms.

The information in this section focuses on three main software characteristics required to support this new paradigm in communications:

- High availability, high reliability features inherent in the RTOS are critical to serve as a solid foundation for network processor based communications equipment.
- Integrated RTOS extensions that effectively support

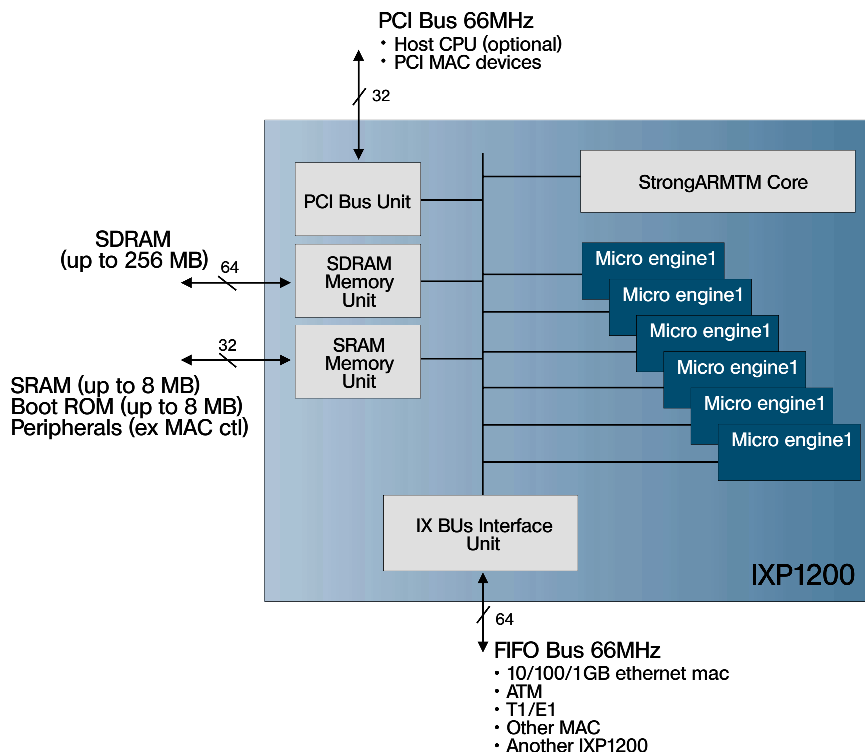

Figure 2. The IXP1200 Network Processor from Intel.

Copyright 2001 by Dedicated Systems Magazine - 2001 Q2 (http://www.dedicated-systems.com)

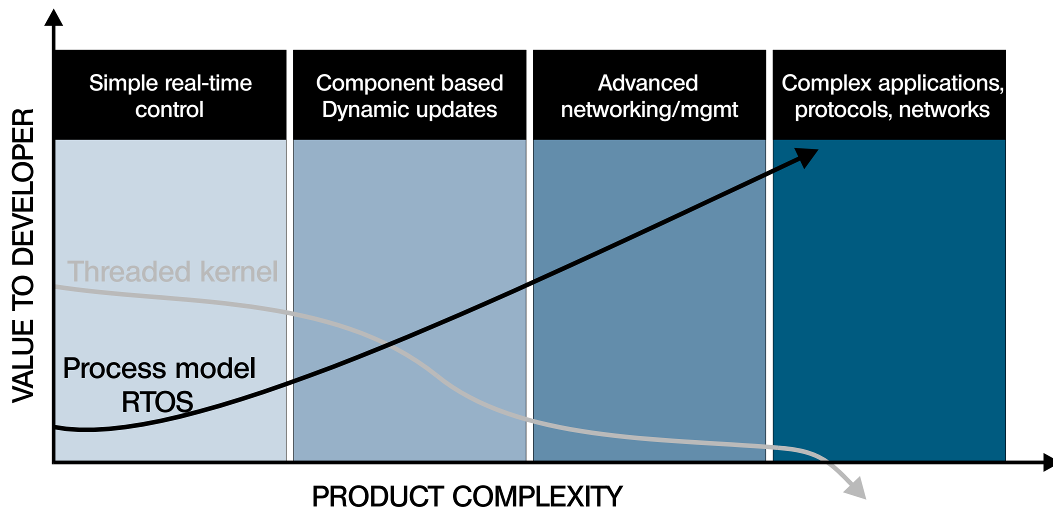

Figure 3. Threaded and process model value as complexity increases.

communications middleware such as protocol stacks and network management software are critical software pieces for network processors.

- Microcode experience and solutions are the great unknown. How easy is it to turn your ASIC designers into microcode writers? Are there places to get off the shelf microcode solutions and/or components? How about consulting services for microcode "built to spec"? All these questions point to a need for a network processor RTOS to have an answer that accommodates and even provides microcode functionality, enabling designers to focus on product differentiation.

Inherent RTOS Features

In the past 15 years, I've talked to hundreds of communications companies that use a variety of RTOS solutions. These solutions fall generally in to two camps: threads based and process model. Threaded kernels are simplistic in nature, offering task switching, interrupt processing and inter-thread communications facilities. Process model operating systems incorporate memory management unit (MMU) support which traps application software from accidentally or maliciously corrupting any data areas or system resources not granted to the application by the operating system.

Figure 3 shows that as product complexity increases, the value of a process model operating system increases. Conversely, as complexity increases, a threads based kernel becomes more problematic in terms of testing, integration, and resulting reliability. Threaded environments offer no inherent protection mechanisms, making debugging and validating correctness incredibly difficult in complex products.

The communications industry started as simple real-time control and therefore initially used threaded kernels. As time passed, communications complexity increased, even within telephony equipment. Technologies like ISDN, X.25, SNMP, and now ATM and internet protocols drive communications complexity up to a level far beyond being able to guarantee reliability with a threaded kernel. So why have communications designers continued to use fundamentally flawed kernels for these applications? Legacy code. So much code has had to be developed in these unprotected environments, the costs associated with moving forward with a more appropriate process model solution makes the transition difficult. Slowly, over the last few years, large communications manufacturers have transitioned from threads based kernels to process model operating systems. Unfortunately, it usually takes a near emergency experience before manufacturers will admit that this transition is necessary. Network processor technology provides the opportunity to choose a more appropriate software solution with the ability take advantage of the unique characteristics of network processors.

High reliability and security issues

Foremost on communications developers' minds today is high reliability and security in communications equipment. The multi-protocol, multi-network communications world we live in today mandates designers to buy functional software pieces from various sources. In addition, there is a significant amount of software development done internally to optimize performance, obtain statistics, and interoperate with other networking equipment. The result is a software environment that blends purchased code with internally written code on top of a RTOS ported to the circuit board. This kind of software environment has caused practical engineers to shift focus from "we'll just write correct code and not worry about it" to "what happens when a bug or failure manifests itself?"

There are three main requirements to ensure reliability and security for a communications system - application-application protection, application-system protection, accidental or malicious corruption of executables.

Process model operating systems provide a high level of protection between applications. All memory and I/O resources are kept track of by the operating system and any attempt to accidentally or maliciously work

 Copyright 2001 by Dedicated Systems Magazine - 2001 Q2 (http://www.dedicated-systems.com)

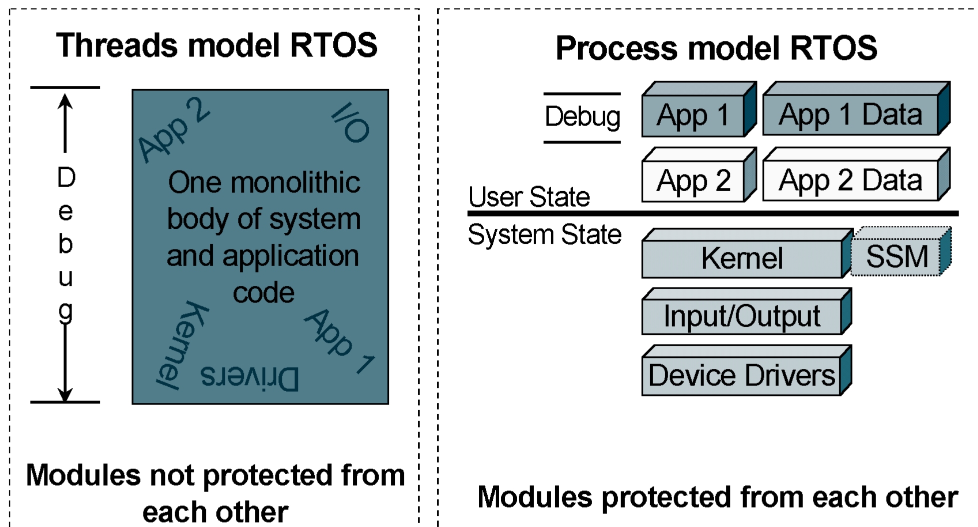

Figure 4. Threads versus process model sofware architectures.

outside the bounds causes graceful exit and cleanup of the offending process by the operating system. Threads model kernels have no inherent protection. Some threads based kernel vendors have developed MMU libraries where the application can register and check against registered memory areas. This is better than nothing, but it's still voluntary protection and if there are bugs in the MMU usage, all bets are off anyway. One threads based vendor is promoting an environment called ”Protected Domain Technology”. This technology isn't available at the time of this paper, but from papers and presentations, this technology simply allows applications to register access for specific areas of memory, but doesn't address inherent memory and resource protection for unique applications.

Application-system protection is usually provided in the form of a user/system state boundary. All application code runs in user space with specific monitoring by the operating system. Resources are requested from and granted by the operating system through a variety of mechanisms. This way the system designer can test all user-to-system calls to ensure no matter how the application calls the system, it cannot cause an exception - only returned errors.

Finally, executable corruption is an interesting topic. At the time of this paper, the ”Love Bug” and some derivatives have floated around the internet. Estimates say that associated costs are in the billions of dollars to those affected. This is a very real issue and must be addressed at the heart of the overall system software - the operating system. Certainly protection and user/system state boundaries help to an extent. Microware's OS-9 operating system has an additional capability to address security. OS-9 has built-in CRCs for every module in the system. The kernel is configured to only load modules with valid CRCs. With this extra level of protection, traditional attacks by computer viruses (where code is attached to an existing program and the executable entry point is changed within the existing program to point to the ”bad code”) are further protected. When OS-9 detects a bad CRC in a module the kernel will not load or run the code. In addition to the module CRC, OS-9 also contains an edition and revision number inside of every module. This inherent configuration management information enables system administrators to guarantee proper identification of every module in a given product release.

High Availability/Field upgrade

High availability is a major topic in today's networking industry. The ability to ”hot swap” boards in systems and upgrade systems while on-line and in-use is paramount for new communications equipment.

A network processor operating system must allow the ability to dynamically add, remove, and replace individual system components or applications on the fly while the communications system is on-line and in-use. The capability inherent in this kind of communications foundation opens the door to less down time due to maintenance and faster, more frequent upgrades in services.

The key enabler to dynamic upgrading is the basic architecture of the RTOS itself. Every RTOS on the market today claims ”modular” and ”scaleable” to the point where these adjectives are now meaningless in the industry. Many RTOSs that claim modularity refer to the process of blending together multiple libraries at link time. The marketing spin is that each library can be left in or taken out of the linking process, thereby being modular. The fault of this rationale is that in the end, what gets loaded onto the target device is one monolithic blended body of code. These RTOSs are more accurately classified as ”link time modular” and ”runtime monolithic”.

Component based architectures generate individual unit-testable modules (or components). These components are collectively concatenated together to form an image that is loaded on the target system. For example, in the OS-9 architectural diagram in figure 5, each run-time image component is still able to be added, removed, and replaced while the system is on-line and

Copyright 2001 by Dedicated Systems Magazine - 2001 Q2 (http://www.dedicated-systems.com)

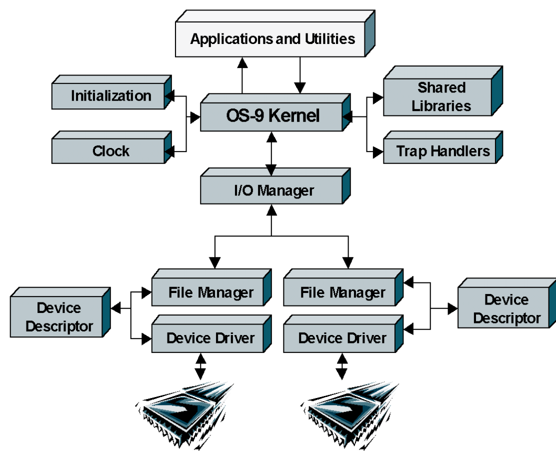

Figure 5. Component based RTOS's like OS-9 are better suited for HA and hot swap.

in-use.

Surprisingly few operating systems on the market have this capability. There are products that allow for field upgrade, but these products are only as capable as the operating system foundation allows them to be. Be wary of the marketing spin - when you ask if an RTOS is component based, make sure you follow up with questions like "and what defines each individual component in the run time image? Does each component have a defined module format? CRCs? Headers? How does the kernel support individual component addition, removal and replacement?"

Fault Resilience

The more commercial product software is incorporated with internally developed software, the more important fault resilience becomes. As communications devices increase in complexity, it becomes necessary for the real-time operating system foundation to have robust exception handling facilities in order to report exceptions and keep the main system alive during exception analysis.

The key requirement here is for the operating system baseline to have a standard exception handling procedure in addition to having an exception framework that allows custom exception handling routines to be invoked.

The standard exception handling routine must return all memory and resources granted to the application and terminate the application with a descriptive error code. In addition, the operating system should allow custom exception handlers to be inserted for handling specific exception conditions. For instance, if there is a mis-aligned data reference, an exception handler could be created to analyze the problem and either provide descriptive information for analysis and/or best attempt recovery procedures to keep the system completely operational. This kind of inherent exception handling capability allows a higher level of instrumentation and automatic recovery for new age communications devices.

Communications Extensions Integrated with the RTOS

Now that we've covered key inherent properties within an RTOS for network processor applications, let's look at the next layer - how accommodating is the RTOS for snapping-in communications capability.

Figure 6 illustrates the traditional approach to implementing protocols for an RTOS. There is usually one way to implement anything on an RTOS - make a task. How do you incorporate protocols? Make them tasks.

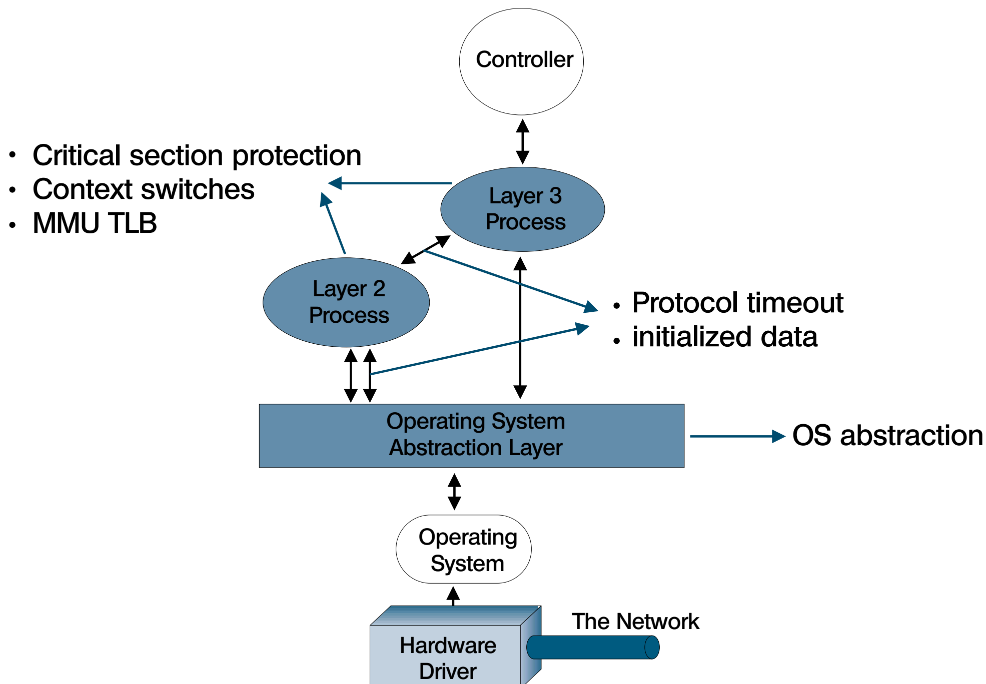

Figure 6. Traditional task-based approach to implementing protocol stacks.

Copyright 2001 by Dedicated Systems Magazine - 2001 Q2 (http://www.dedicated-systems.com)

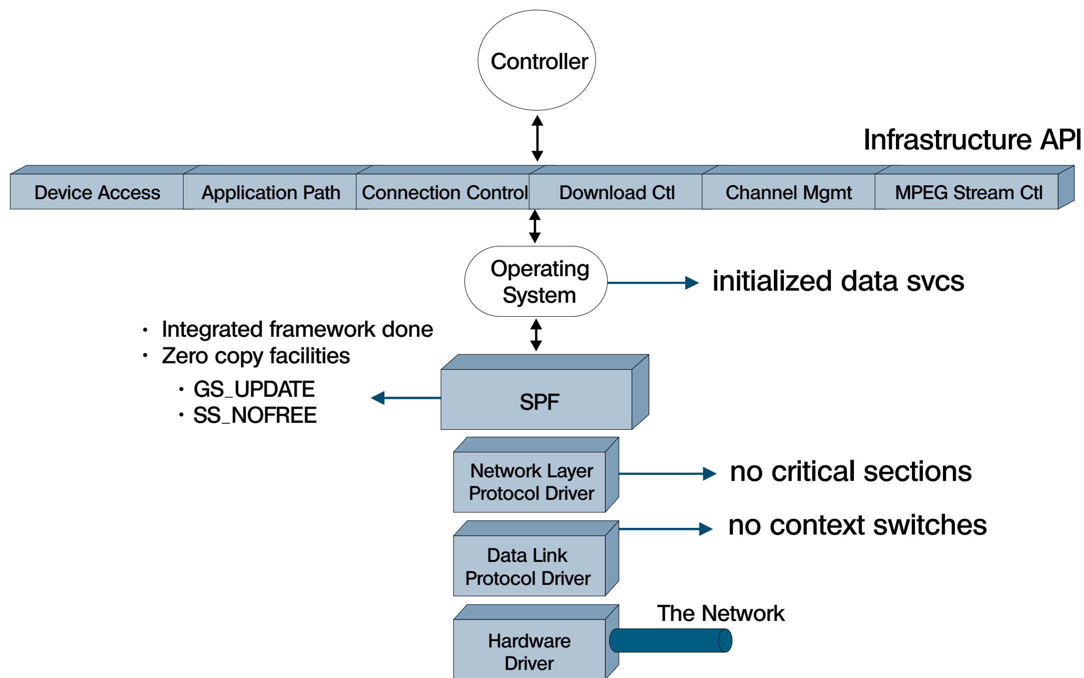

Figure 7. Driver based communications framework.

Network management? Tasks. Write your application? Tasks. In order to maintain some logical separation, each protocol layer is implemented as an individual task. Since there is no alternative accommodation by the RTOS, an operating system abstraction layer is developed starting with the generic RTOS services and builds services required by communications protocol stacks.

There are inefficiencies associated with this approach:

- As data and control flows through the system, there are multiple context switches occurring in the software, which restricts the CPU time processing packets and increases the overall system overhead.
- Critical sections, typically using semaphores, also represent overhead when passing data and control between tasks.
- The RTOS only exposes generic services to the application environment. If the RTOS abstraction layer writer doesn't have intimate knowledge of the RTOS, the abstraction layer implementation tends to be inefficient. In addition, there may be hidden services within the RTOS that could be taken advantage of if the RTOS was tuned for communications stack processing.

With data rates from 9600 baud to 10Mbps, these inefficiencies tend to be small when compared with the data rates, so the inefficiencies were tolerated. However, with fast ethernet data rates and above, these inefficiencies become intolerable. While most of the packet processing in network processor environments will be done by the microcode engines, that doesn't mean it's allowable for the core processor to handle configuration and management packets inefficiently. While there will be fewer packets to process, the line rates are extremely high. In times of significant reconfiguration or network management activities, there will still be significant load on the core processor.

Figure 7 shows a new driver-based approach to protocol stack design. This approach involves extending the RTOS to integrate a communications software framework and expose services that are critical to high performance communications activities. The result is a real-time operating system with a built-in, high performance communications framework.

The benefits of this approach include:

- **Driver based** design eliminates context switching and critical section protection, providing up to 30% performance boost attributed to the framework alone[4].
- **Multi-vendor interoperability.** Since the framework defines the control and data interface between each protocol layer, its possible for multiple vendors to implement protocols to the specification that are automatically interoperable.
- **RTOS integration with microengines.** Not only does the driver based approach provide performance advantage, the specified protocol layer interface allows a microcode interface component to be implemented on the core processor once, providing a graceful interface extension for processing done on the core processor and processing in the microengines.

Driver based communications frameworks do exist in product form. Microware's SoftStax™ communications framework was developed for the OS-9 operating system as a result of stumbling over the problems associated with a task based approach when developing network interfaces for set top boxes running DS-3 and ATM data rates in the mid 1990's. Network processors leverage SoftStax technology even more. First, the

microengines need some kind of standardized interface between the microcode and core processor boundary. Second, more complicated packet processing (such as protocol conversion) may still be handled by the core processor due to space limitations. SoftStax optimized processing of packets by the core processor. Many articles have also been written and have been included in the reference section of this paper.

Microcode Experience and Solutions

Historical parallels between microengines and microprocessors, microcode and RTOSs are striking. The advent of the microprocessor started a software industry that advanced from switches and punch cards to compilers, debuggers, and operating systems. Microcode is another example of silicon capability that will necessitate similar software technologies.

The IXP1200 is a good example of rapid advancement of microengine software technology. The IXP1200 comes with a microengine compiler and development tool suit to simulate threads of execution running on the microcode engines while on a PC. Natural evolution in this environment is some form of communications software foundation for microengines that makes it easier to develop microcode.

The IXP1200 compiler allocates three classes of registers based on naming conventions. An alphanumeric name is allocated for general purpose register access, name preceded by a '$' indicates a scratchpad register out of SRAM, and a name preceded by a '$$' refers to a register allocated from SDRAM. There is no need to define registers as variables - the compiler automatically allocates them as they are used. Instruction format is as follows:

```
‹address name›[operation, register_name, base, offset, number of memory units]
```

for example, the following instruction says to read the 16 bytes (4 words) of memory from the SRAM register at base address of sram_base and offset of sram_offset into SRAM.

```
sram[read, $sram_register, sram_base, sram_offset, 4]
```

There are a few other instructions to go with this general instruction format. This programming environment more closely resembles assembly coding than ASIC equations. This is where software experience with microcode can be a valuable addition to the network processor operating system portfolio. Network processor paradigms are centered around microengines. Therefore, it stands to reason that a network processor addresses the microcode interface and solution set for a complete solution.

Putting it all together

Now that we've looked at the entire picture for a network processor and the software baseline, let's take a look at one configuration involving the OS-9 operating system and the IXP1200 network processor. Figure 8 shows the software block diagram of an OS-9 based IXP1200 solution for a fast ethernet/ATM bridge/router/gateway.

Notice in the diagram, there are two functional parts to the solution - the software residing on the StrongARM core processor within the IXP1200 (OS-9 for Embedded Systems) and the software running on the microengines (Microcode solutions library).

The software residing on the core processor is the OS-9 RTOS and SoftStax communications framework which includes the block labeled "SPF" and all the blocks below. This design is able to accommodate straight 10/100/1000BaseT, PPP over ethernet, serial, or ATM, and ATM signaling and network management. Network management is also available in the form of SNMP. The design also includes a serial port for a console interface and a flash file system that provides a file system and automated wear leveling algorithms to enable long life for the flash parts. The serial driver block may reside as a device driver in the core processor software as shown, or as a driver interface to a serial port accessed through the microengines.

This design really highlights the value of an integrated communications framework. The ATM signaling stack (labeled Q.2931, Q.2130, Q.2110) and ATM management entity (labeled ILMI) are available as a pre-integrated plug-in to the SoftStax environment. This kind of pre-integration can shrink a 3-6 month integration effort into a simple installation exercise.

At the bottom piece of the SoftStax architecture is the interface component to the microcode engines, labeled "Microcode interface" in the diagram. A defined interface consisting of the "SoftStax interface" and the microcode interface enable this control and data passing software boundary to be reused regardless of the communications protocols or topologies involved.

The microcode solutions consist of soft-SAR for ATM (SAR block), IP over ATM functionality (IPoA block), and two IX Bus interface blocks that send and receive data over their respective network topologies. The number of interfaces supported is a function of the speed of the microengines.

Incoming data flows through the device from one of the interface blocks (OC-3 or MAC). The data packets are then processed by the microcode. The microcode utilizes the information in the configuration table to classify the packet and route the packet out the appropriate interface, adding headers or reformatting the packet as needed per IP over ATM processing. If the packet is classified as being interesting to the core processor software, the packet will pass through the microcode interface. At this point, standard packet processing on the StrongARM core will occur. If the packet is a network management query, it will be passed to the SNMP agent. The SNMP agent will then respond with the appropriate response through the TCP/IP stack and out the appropriate interface via the microcode. Packets may also come in that cause changes by the configuration and provisioning application to the configuration table. The result of this re-routing information will be used in subsequent lookups by the microcode.

Copyright 2001 by Dedicated Systems Magazine - 2001 Q2 (http://www.dedicated-systems.com)

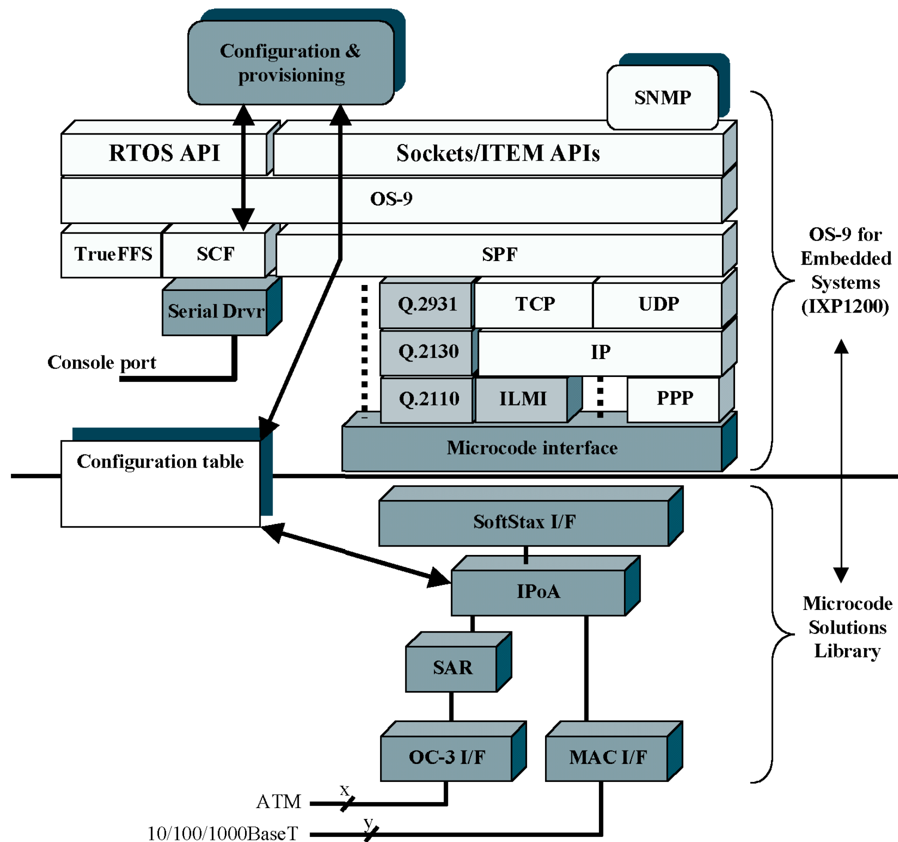

Figure 8. Software block diagram example of an IXP1200 based system.

SUMMARY

Past performance tells us that we can expect silicon performance and capability to continually advance. Processors with increased performance, integrated capabilities have continued to advance at a fast pace. Now we find ourselves at the beginning of a network processor era.

I have yet to hear the same accolades going out to the software industry. Software development cycles are not being accelerated. Over the past ten years, traditionally used threads based kernels have not evolved to accommodate these rapid silicon changes. Many popular RTOSs are fundamentally flawed for communications applications. Communications software frameworks are an absolutely vital element to taking advantage of network processors, yet few vendors provide this kind of software integration.

In this paper we looked at a short history of both communications silicon and software. Key properties inherent in the RTOS foundation were described. We talked about network processor technology and the strengths and challenges this technology presents to the software community. Finally, we looked at a specific network processor application from a software block diagram point of view and described functionality and general data flow through the system.

Paradigm shifts are always a scary proposition for the affected industry. Companies successful with the current paradigm often close their eyes to the change, discounting it as being ”unfeasable”, ”no real value”, or ”too costly”. Companies that attempt to apply current software solutions to network processor technology are destined to failure without evaluating software solutions that enhance this new paradigm and accelerate its development. If you don’t, chances are your competition is.

REFERENCES

1. Weiss, Ray (2000) Network Silicon Drives Next Generation Telecom & Datacom RTC Magazine, Volume IX, Number 1, January 2000
2. Cravotta, Nicholas (1999) Network Processors: The Sky’s the limit EDN, 11/24/99
3. Schwaderer, Curtis (1999) Developing Custom Network Protocols Dr. Dobbs Journal, #303, September 1999 ■

ACurtis Schwaderer is the Director of Network Technologies for Microware Systems Corporation. Curt has a masters degree in Computer Engineering from Iowa State University with an emphasis on networked embedded systems and network security. With over 12 years of networking experience, Curtis also designed and implemented secure/non-secure digital switching systems for E-Systems/Raytheon. Curt also has a patent pending in the communications industry.