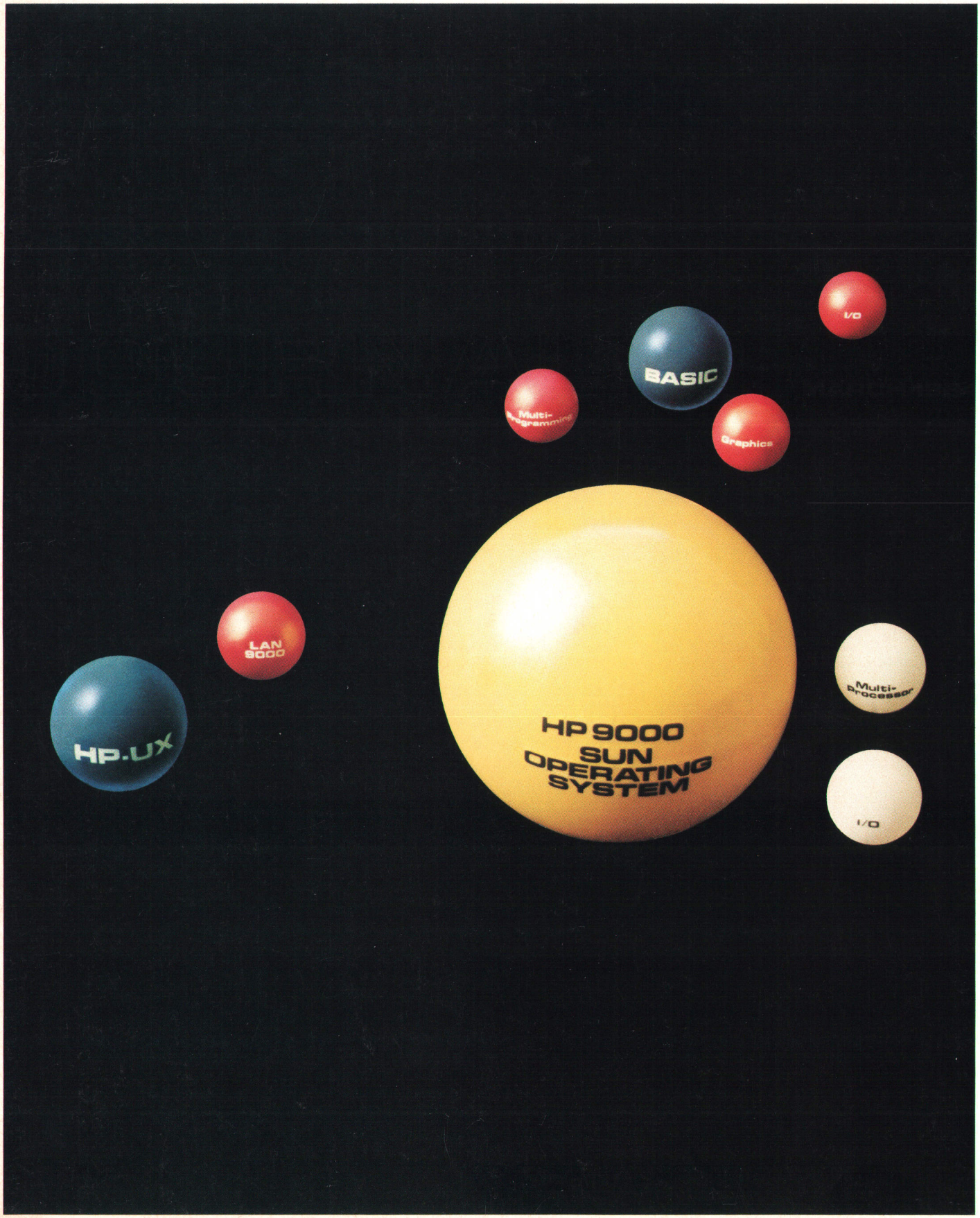
HEWLETT-PACKARD
JOURNAL

MARCH 1984

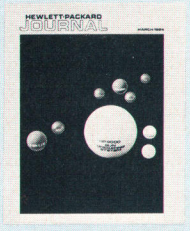


HEWLETT·PACKARD JOURNAL
Technical Information from the Laboratories of Hewlett-Packard Company

MARCH 1984 Volume 35 • Number 3

Contents:

In this Issue:



The solar system on this month's cover represents the system software for the HP 9000 Series 500 Computers. We first told you about the HP 9000 in our August 1983 issue, which was devoted to the advanced technology that makes the Series 500 possible. You may recall reading about the HP 9000's five VLSI (very large-scale integration) chips—among them a 32-bit, 450,000-transistor central processor chip—made by a high-tech integrated circuit process called NMOS III. To help manage the heat generated by all those densely packed circuits, a new kind of circuit board, called a finstrate, was developed. The finstrates in each HP 9000 Series 500 Computer are contained in a lunchpail-sized module that holds up to three central processors. These technological developments make it possible to put on an engineer's desk a computer that has more power than some mainframe computers—"mainframe" being the name applied only to the largest computers. The HP 9000 Model 520 is the desktop mainframe. Models 530 and 540 are, respectively, rack-mount and cabinet versions designed to serve multiple users.

Although the lunchpail-sized module is the beginning, between it and that desktop mainframe is a great deal of development in both hardware and software. This month's issue covers the system software development. In May, we'll cover the hardware development, and in future issues, we'll carry articles on significant applications software packages. In this issue you can read about operating systems, languages, input/output, networking, and multiprocessor management. An unusual aspect of the HP 9000 Series 500 is that there are two levels of operating system. What the user sees is either an advanced version of the HP BASIC system or an HP version of Bell Laboratories' UNIX operating system. Underlying those systems is the Series 500's SUN operating system, whose name gave us the idea for our cover photo. The SUN concept proved invaluable in the development of the two user operating systems.

-R. P. Dolan

A New 32-Bit VLSI Computer Family: Part II—Software

Based on HP's proprietary 32-bit VLSI NMOS-III technology, the HP 9000 Series 500 Computers use local area networking and HP-UX, HP's enhanced version of UNIX™. An advanced version of BASIC that uses run-time compiling is available on the Model 520 integrated workstation.

by Michael V. Hetrick and Michael L. Kolesar

IN 1981 HEWLETT-PACKARD described the development of a single-chip 32-bit processor[1] fabricated with a new VLSI process technology called NMOS III.[2] This new technology was also used to develop four other 32-bit chips that, coupled with the design of a special copper-cored circuit board called a finstrate, enable a powerful multiprocessor 32-bit computer system to be packaged within a module no larger than a loaf of bread. The design of the five chips, the module, and the finstrate, and the NMOS-III process were discussed in last August's issue.

The compact module, called the Memory/Processor Module, forms the heart of a desktop engineering computer workstation, the HP 9000 Computer, introduced by HP in 1983. Now known as the HP 9000 Model 520, it contains a 5¼-inch flexible disc drive, has four I/O slots, and has a choice of either a color or monochromatic 13-inch CRT display. Depending on the choice of the twelve finstrates possible in the Memory/Processor Module, up to three CPUs, three I/O processors, or 2.5M bytes of RAM can be installed. Available options include an internal thermal printer and an internal 10M-byte hard disc memory.

A sophisticated internal operating system, called SUN, was developed to coordinate this compact multiprocessor computer system. A high-performance interactive high-level language system is required to allow a user to take full advantage of the features included in the Model 520. An enhanced version of BASIC and a run-time compiling technique were developed. This version was designed as a superset of the BASIC used on earlier HP desktop computers so that users could easily port existing software to the Model 520.

In late 1983, the HP 9000 family of computers was defined to include some earlier 16-bit technical desktop computers,[3] now known as the Series 200, and alternative packages for the Memory/Processor Module, which with the Model 520, form the Series 500. To simplify the porting of software developed by other companies to the HP 9000 family, HP-UX, an enhanced version of UNIX™, was developed. LAN 9000 was developed to provide local area networking.

Fig. 1 depicts all current HP 9000 models. The primary distinction between the Series 200 and Series 500 Computers is in their microprocessor, or central processing unit (CPU), and the ensuing system design. All Series 200 models are based on Motorola's 16/32-bit 68000 microprocessor, while all Series 500 models use HP's proprietary 32-bit

UNIX is a U.S. trademark of Bell Laboratories.

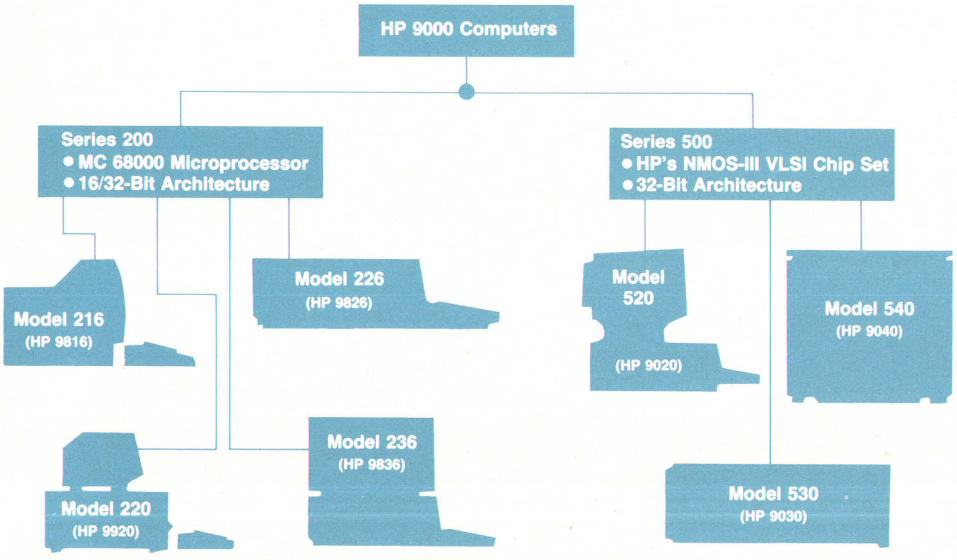


Fig. 1. *The HP 9000 family of computers includes the Series 200 and the Series 500 Computers. The Series 200 is based on the 16/32-bit 68000 microprocessor and the Series 500 is based on HP's proprietary 32-bit VLSI NMOS-III chip set. The Series 200 is lower in cost and the Series 500 has higher performance. Programs developed in BASIC on the Series 200 can be ported to the Model 520 of the Series 500 for decreased computation time and other performance advantages.*

Contrasting Project Management

The relative magnitudes of the BASIC and HP-UX projects created interesting and contrasting software management techniques. BASIC, being relatively small in code size by today's standards (less than one megabyte) was implemented entirely within HP's Fort Collins Systems Division (FSD). A development environment known as MODCAL, which ran on previous-generation desktop computers, was an effective tool for code development. Since the project was executed by a few engineering groups in one department, coordination among the design team was extremely efficient.

HP-UX development, on the other hand, was much larger in scope; it resulted in approximately 10 megabytes of system code. Features such as multiple languages, graphics, networking, and fundamental program development tools were required. HP entities outside of FSD had the expertise to contribute in some of these key areas. Thus, two California organizations—the Computer Language Laboratory (CLL) and the Engineering Productivity Division (EPD)—along with the Colorado Networks Operation (CNO) provided FSD with major software subsystems. CLL produced the FORTRAN and Pascal compilers, EPD developed HP's well-known two- and three-dimensional graphics libraries, and CNO provided most of the data communications and networking software.

FSD ported the UNIX™ commands and created the System III UNIX interface to the existing Series 500 operating system. FSD was also responsible for coordinating the entire software development and integrating all subsystems into a cohesive product.

As the HP-UX asynchronous communications software became functional, it was used to transmit messages and internal software updates between divisions. FSD and CNO capitalized on high-speed local area network prototype hardware and software to update local systems electronically.

Thus, many HP-UX software subsystems became key development tools even as they were being created. (The UNIX command set, C, FORTRAN, and Pascal compilers, and SCCS, the source code control system, are additional examples.) Their everyday use not only contributed to our development productivity, but also served as a prime example of how internal use of what will become a product improves the product's overall quality in a way that is not otherwise possible.

*UNIX is a U.S. trademark of Bell Laboratories

-Michael V. Hetrick
-Michael L. Kolesar

CPU chip set mentioned earlier. Highly compatible system software spans the lower-cost Series 200 and higher-performance Series 500 workstations to provide a broad price/performance product family. BASIC is offered on all but the Model 530 and Model 540; HP-UX is offered on all but the Model 216 and Model 226. A standard HP Pascal development environment also appears on all Series 200 models.

This issue discusses the development of the Series 500 software systems with the exception of the multitasking, graphics, and I/O subsystems for Model 520 BASIC. They will be discussed with the hardware design of the Series 500 in the May issue, which will conclude the story of the development of the HP 9000 Series 500 Computers.

Software Organization

The software for the Series 500 is modular and is easily decomposed into smaller building blocks. The design kept the internal SUN operating system clearly distinct from the code for the BASIC language. This separation was necessary to provide the foundation for a true multilingual system. For example, we knew that the Series 500 with virtual memory would be an excellent FORTRAN engine.

The separation or layering of the software made it necessary to define a powerful and flexible set of operating system entry points to support the real-time event-driven BASIC language needs. This set of underpinnings also serves as the basis for the HP-UX system.

The SUN operating system hides most of the hardware details from the higher-level subsystems. The initial version of SUN supports the Model 520's demanding 700-keyword BASIC language with its new run-time compiler. The support for multiple CPUs and I/O processors was designed in from the beginning. The BASIC language system supports memory-resident programs and data, but does not support virtual memory. Besides the BASIC language mainframe code, several option packages extend its capabilities by adding two- and three-dimensional color graphics, HP's IMAGE data base management and query system, extended I/O, extended mass storage with multiple disc formats, multitasking, advanced programming such as matrix manipulations, and I/O drivers for a variety of interfaces and devices. BASIC's highly integrated human interface causes it to be provided only on the Model 520, the integrated desktop version of the Series 500. Special hardware in the Model 520's display unit is used to achieve excellent performance for BASIC's text and graphics window facilities. The Model 520 keyboard contains special control keys used in BASIC, such as **RUN**, **STOP**, **STEP**, and **PAUSE**, in addition to the keys normally found on a terminal keyboard.

We chose UNIX as the best available environment to support FORTRAN and other standard languages. A second version of the SUN operating system was built using the modules from the first version, but with the important virtual memory feature added. The diagrams in Fig. 2 and Fig. 3 show the similarity of the BASIC and HP-UX systems. This leverage paid off handsomely, because almost from the beginning, the terminals, discs and printers worked reliably

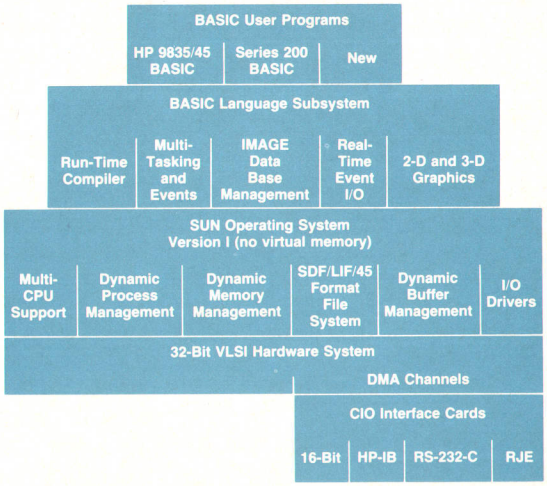


Fig. 2. *Block diagram of the BASIC language system for the HP 9000 Model 520 Computer.*

for HP-UX. Also, the real-time and multiprocessor design carried over to HP-UX to give it a more solid basis for performance extensions, device I/O, and real-time than could be achieved with a ported system.

A thin layer of code maps the HP-UX intrinsic calls into the underlying SUN intrinsics. The same HP Structured Directory Format (SDF) hierarchical file system used by BASIC is also used by HP-UX. It is almost indistinguishable from the System III UNIX file system except that it is more reliable and less susceptible to corruption from power failures and system crashes. This layered design was a concern because it could lead to deviations from the UNIX semantics defined by Bell Laboratories. Therefore, a set of extensive and comprehensive kernel test programs was devised to determine if any detectable differences had been introduced. With only a small additional effort, the layered kernel passed the validation tests. The same set of test programs is being used to verify the Series 200 HP-UX kernel and future releases of the Series 500 kernel.

The commands and libraries offered are selected from both the Bell Laboratories and the University of California at Berkeley versions of UNIX. Those most needed for program transport and development are included. The three user program languages offered are C, Pascal, and FORTRAN 77. The calling sequences allow mixing of languages at the subroutine level and the sharing of all library routines.

The definition of HP-UX includes not just compatibility with Bell Laboratories' UNIX System III, but also HP extensions. The current system offers IMAGE data base management, AGP/DGL graphics, and local area networking based on ARPANET TCP/IP and Ethernet protocols with both file and process services.

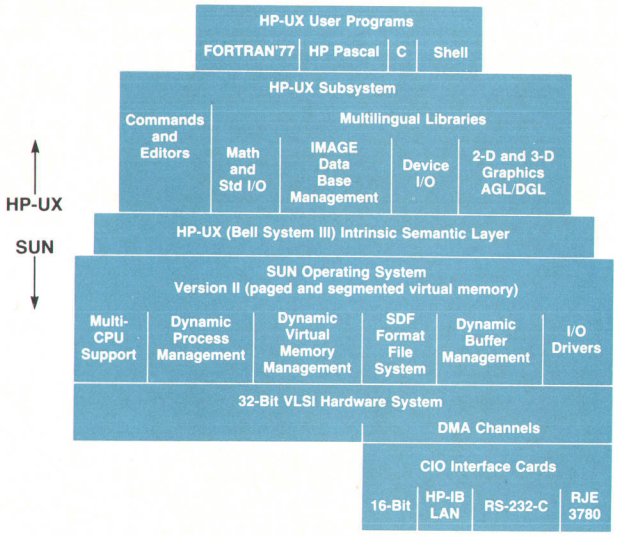


Fig. 3. *Block diagram of the HP-UX operating system for the HP 9000 Series 500 Computers.*

References

1. J. Beyers, et al, "A 32b VLSI Chip," *Digest of Technical Papers*, 1981 IEEE International Solid-State Circuits Conference, THAM 9.1.
2. J. Mikkelson, et al, "An NMOS VLSI Process for Fabrication of a 32b CPU Chip," *ibid*, THAM 9.2.
3. *Hewlett-Packard Journal*, Vol. 33, no. 5, May 1982.

The Development of a BASIC Language Subsystem

The development of the BASIC language subsystem for the HP 9000 Model 520 Computer had one primary goal—to allow this new workstation to be a complete functional replacement for the HP 9835 and HP 9845 Computers[1] while achieving at least ten times the performance of the HP 9845 by using HP's new 32-bit NMOS-III custom VLSI chip set.[2] In addition, new features were needed to keep pace with the new applications that such a capable machine would encompass. The development schedule for this top-of-the-line BASIC machine ran in parallel with the chip set development.

The design team decomposed these high-level goals into many challenging technical goals, uppermost being to provide a growth path for HP's current BASIC language customers through a high degree of compatibility, and to add new functions suited to the power of the new hardware. The BASIC language was unified and extended in cooperation with the Series 200 Computers[3] language team while retaining a high degree of program compatibility with the earlier HP 9835 and HP 9845. Thus, programs from either generation of machines move easily onto the Model 520. Almost no differences are notable between Series 200 BASIC and Model 520 BASIC. Even though most HP 9845 statements are retained, the uniformity of the evolving language dictated that some differences would result. An optional translator for HP 9845 programs to achieve a more precise semantic match is available.

The major technical contribution to support the performance goals was a run-time compiler for the BASIC language. This compiler appears to the programmer to be the same as our traditional interpretive environment, preserving such features as tracing, line stepping, execution of statements from the keyboard and manipulation of a running program's variables. A running program can be paused, lines can be added, deleted, or modified, and execution then continued from its point of suspension. All of these features are still supported even though the user's program is not interpreted, but compiled into object code and directly executed.

The resulting high-productivity programming environment uses execution modes and trap instructions built into the new processor. Parallel chip set and software development allowed many specialized instructions to be added to the processor in support of these interactive features as well as the language itself. These included some of the traps, the string manipulation set, and bit manipulation. The resulting BASIC language system has over 700 keywords that encompass significant data base management, graphics and I/O capabilities.

The new compiled environment retains the event-driven, real-time program control of the HP 9845 and HP 9000 Model 226 Computers. Program branching and flow are tied to both hardware and software events through the use of **ON** statements. These statements define the asynchronous branching that is to occur when selected events happens.

Also in support of the performance objectives, the new machine uses the emerging IEEE binary floating-point mathematics standard instead of the traditional decimal mathematics. In addition to making use of the fast microcoded floating-point on the microprocessor, binary mathematics is used in new algorithms for

computing transcendental and other functions which are both faster and more accurate.

Multitasking was added to improve the user's access to the machine and to use the improved processor power. Simultaneous program execution and development are now supported. Up to 60 user processes can be run simultaneously. These processes share the system resources and peripherals. For communciations and synchronization they have "named-event" signaling. Memory resident volumes and files were added for rapid shared-data access. File locking allows for atomic (indivisible) update of a shared file.

The system architecture provides for multiple identical CPUs. This feature was incorporated into the operating system so that the power of many processors could be directed at runable tasks. The fundamental design supports these multiple CPUs as homogeneous and anonymous computing resources. Since all CPUs have symmetric access to all I/O, there is no need to introduce master-slave relationships. The only externally visible effect of adding more CPUs is increased throughput.

Full-screen editing and multiple user-defined windows in both graphics and alpha displays were added to improve the human interface. Both public and private windows are supported, including arbitrary window overlap and last update priority display. These windows act much like sheets of paper on your desk, with the topmost sheets occluding the sheets below where they overlap. The window structure is dynamic—even the system message areas can be relocated anywhere on the screen.

An example of new hardware capability that mapped into a new set of language features is the internal, nonvolatile, real-time clock, which facilitates using time to schedule program events. The new file system also uses the clock to time-stamp files as they are created or changed and the lister dates hard copy.

The graphics definition was extended to support multiple input devices with tracking and event capture, and the transformation pipeline includes both two- and three-dimensional modeling modes.

Development Tools

A very accurate emulation program that mimicked the execution of the new machine's instruction set was written for execution on the distributed HP 9845 workstations. This software emulation was so accurate that it took only ten minutes from the time the first chip set was delivered to the software team until the system was up and running a BASIC program in compiled code on the new Model 520 hardware.

The instruction set[4] of the new VLSI CPU chip provided the hooks for a highly interactive symbolic debugging tool. This debugger provided a simple transition between running and stepping of systems programs. Procedure, line, and assembly level stepping are selected on the fly. Program flow is displayed symbolically at the appropriate level. Variables can be referenced symbolically.

Pascal was chosen as the systems programming language with extensions for separate modular compilation to support large team program development. This new language is called MODCAL for MODular PasCAL. MODCAL is very similar to Wirth's Modula II, but was designed independently by HP. For the programming environment, the UCSD (University of California at San Diego) p system was selected because it was written in Pascal, was easily ported to the HP 9845, and had the other tools, such as editors, that were needed.

Another important decision was to separate the BASIC language from its operating system support. Clear separation of the operating system provided code that could be leveraged for the HP-UX project and reduced the cost of maintaining the Model 520 BASIC system. The underpinnings for the HP 9000 multiple-CPU HP-UX system are the same operating system modules used for BASIC.

Over 750K bytes of code are in the BASIC system for the Model 520, forming the most powerful program development environment ever provided for an HP desktop computer system. The resulting system's speed, graphics, and real-time, event-driven I/O capability make it a very powerful engineering tool. A large number of HP 9845 and Model 226 programs have been easily moved onto the Model 520. Performance gains average 50 to 100 times the performance of the HP 9845B Computer and more than 10 times the performance of the HP 9845, option 200, for computation-limited tasks. The same computational tasks average 15 times faster than on the BASIC version of the Model 226 Computer.

Acknowledgments

Chris Christopher was the lab manager and Bill Eads was the section manager and temporarily managed the graphics team. Jeff Eastman developed MODCAL, managed the initial graphics team, and chose the CORE style pipeline. Scott Wang managed the tools team responsible for the interactive debugger and MODCAL optimizations, Dave Maitland was responsible for early project management of the design team, Denny Georg and Dan Osecky were the principal operating system designers and the implementors of the emulation tool, Dave Landers made the runtime compiler a reality, and Dan Osecky was the primary architect of the operating system support for multiple CPUs.

References

1. *Hewlett-Packard Journal*, Vol. 29, no. 8, April 1978.
2. J.W. Beyers, E.R. Zeller, and S.D. Seccombe, "VLSI Technology Packs 32-Bit Computer System into a Small Package," *Hewlett-Packard Journal*, Vol. 34, no. 8, August 1983.
3. *Hewlett-Packard Journal*, Vol. 33, no. 5, May 1982.
4. J.G. Fiasconaro, "Instruction Set for a Single-Chip 32-Bit Processor," *Hewlett-Packard Journal*, Vol. 34, no. 8, August 1983.

-Michael L. Kolesar
-Jack D. Cooley

HP-UX: Implementation of UNIX on the HP 9000 Series 500 Computer Systems

by Scott W. Y. Wang and Jeff B. Lindberg

AN IMPLEMENTATION of the UNIX™ operating system kernel has been layered on top of an existing operating system kernel for the HP 9000 Series 500 Computer Systems. The mapping of UNIX functional requirements onto the capabilities of the underlying operating system is discussed in this article, along with the implementation of UNIX commands and libraries. These pieces of UNIX, along with other extensions added by HP, make up the HP-UX operating system.

The HP-UX operating system is compatible with Bell Laboratories' System III UNIX, and supports most of the standard UNIX commands and libraries. A number of extensions are available, including

- FORTRAN 77
- HP Pascal
- C
- HP's AGP three-dimensional and DGL two-dimensional graphics subroutines
- LAN 9000, an Ethernet-compatible 10M-bit/s local area network
- The vi visual editor
- Virtual memory
- Shared memory
- HP's IMAGE data base management system
- Support of symmetric multiple CPUs.

HP-UX Operating Environment

There are three levels of software in a UNIX system: commands, libraries, and kernel intrinsics (Fig. 1). The commands are user-level programs which can call libraries or kernel intrinsics. Some commands are provided with the operating system as standard utilities. One example is the command interpreter, or shell. Commands can also be written as normal user programs by the user. Libraries are also user-level code, but can be called only from a programming language such as FORTRAN or C. Kernel intrinsics can be called (normally as functions) from user programs or libraries, and provide a fundamental set of operating system operations.

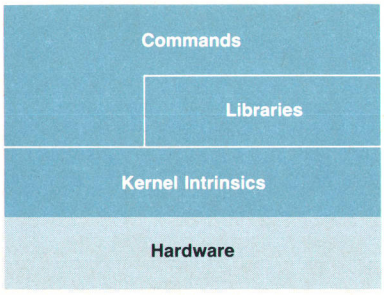


Fig. 1. *UNIX consists of three levels of software—commands, libraries, and kernel intrinsics.*

UNIX Kernel Overview

A standard UNIX kernel provides support for I/O, file system access, process management, real-time clock access, memory allocation, etc. The set of kernel intrinsics is fairly small and simple; only basic operations are supported by the kernel. For example, file manipulation operations such as copying files are done by commands. The command interpreter shell is another capability that is implemented in a user program instead of inside the kernel.

Process Management. The UNIX kernel supports the creation of asynchronous processes that run in the background while the user executes other interactive programs in the foreground. Intrinsics are provided for the creation, termination, and synchronization of processes. Special events

Typical HP-UX Commands

Commands in HP-UX are run by entering the name of the command. For instance, to list the contents of the current working directory, enter ls. This causes a program by that name (which may be located in one of several default directories) to be loaded into memory and to begin executing. Other examples of HP-UX commands are:

cd dirpath	Change the working directory to the directory indicated by dirpath
pwd	Prints the full path name (filename) of the current working directory
vi filename	Invoke the visual editor to edit file filename
rm filename	Remove file filename
cp filename destdir	Copy file filename into directory destdir
cat filename	Print the contents of file filename
cat filename | wc	Print the number of lines, words and characters contained in file filename. wc is the word count command and its input, in this case, is the output of the cat command (due to the pipe created by |).
ls – l	List the contents of the current working directory. The – l is an option that tells the ls command to emit additional information. Most commands accept one or more options.

-Michael L. Connor

are noted by sending signals to one or more processes from other processes or from the kernel. The kernel manages identification fields, such as process ID, user ID, and group ID, which uniquely identify a process or group of processes. The exec intrinsic loads a user program (code and data) into memory from an executable file. The memory model of UNIX is very simple. It consists of the user's program, an execution stack, and a dynamic heap which can be extended or contracted via a kernel intrinsic.

File Manipulation. The UNIX file system is built around a hierarchical directory structure, allowing a directory to contain other directories as well as normal files. Kernel intrinsics are provided to create files, directories, and special files (devices that are in the filename space). The kernel also supports creating and deleting links (alternate names) to files and getting or setting file access modes. A significant feature is the ability to mount a separate disc volume logically onto a directory in an on-line volume. This means that all on-line volumes are part of a single directory hierarchy.

File Access. A single set of I/O intrinsics provides transparent access to files, devices, or the standard input of other processes. A program normally does not know whether its standard input is coming from a file, a device, or another process via an interprocess pipe. Standard operations are provided, including read, write, open, close and status. Special device control is provided via the ioctl intrinsic.

Miscellaneous. Several other features are supported by the standard UNIX kernel, such as real-time clock access, logging accounting information at process termination, and profiling the execution of user programs. The profiling and accounting facilities have not yet been added to HP-UX.

SUN Operating System Kernel

When the HP 9000 project began, the operating system designers took a different approach from that used on HP's previous desktop computers. Even though the first HP 9000 language system was to be an extension of the BASIC language system of the HP 9845 Computer, an objective of the operating system design was to allow other languages in later versions of the product. The system software was designed in a modular, layered fashion (see Figs. 2 and 3 on pages 4 and 5). A central operating system kernel provides a high-level interface to the hardware and machine architecture, while other subsystems provide more specific functions layered on top of this kernel. This operating system kernel, called SUN, is described in detail in the article on page 28.

SUN is written mainly in MODCAL, an enhanced version of Pascal. MODCAL supports information hiding* via modules, an elegant error recovery mechanism, and systems programming extensions such as absolute addressing. A small part of SUN is written in assembly language. The SUN kernel is not visible to the user; instead, it relies on upper-level subsystems such as BASIC or HP-UX to provide a user interface. The major pieces of the SUN operating system kernel handle power-on initialization and memory and process management, and coordinate the file system, drivers, I/O primitives, real-time clock, and interprocess messages.

*Information hiding is a software design approach where the inner workings of an individual section are kept "hidden" from other sections. This allows a section to be changed or updated with minimal concern about its effects on other sections.

An unusual feature of the file and I/O system is the ability to add new directory format structures, device drivers and interface drivers. These modules can be added without affecting the existing SUN kernel code.

Some key pieces are missing from SUN by design, notably the human interface and program loader. The BASIC system provides its own human interface code, which uses the integrated CRT and keyboard of the Model 520, the desktop version of the HP 9000 Series 500 Computers. HP-UX provides a terminal-style human interface to communicate with the user through the integrated CRT and keyboard as well as through normal terminals. HP-UX and BASIC also provide their own program loading facilities.

HP-UX Kernel Strategy

The basic strategy of the HP-UX implementation is to layer the HP-UX kernel definition on top of the SUN kernel. The exact System III UNIX semantics and syntax are kept, but the HP-UX intrinsics are implemented using SUN kernel support instead of porting the Bell Laboratories kernel implementation to the Series 500.

A layer of code called the HP-UX layer resides just above (and in some cases beside) the SUN kernel, as does the BASIC subsystem. However, BASIC and HP-UX are mutually exclusive; only one can be loaded at a time.

The HP-UX layer performs any necessary transformations between UNIX formats and the corresponding SUN formats (e.g., the real-time clock format). It calls procedures in SUN whenever appropriate, but still has full access to the hardware and architecture when needed. The HP-UX layer maintains a number of higher-level data structures to manage HP-UX user processes and user resources.

This layering strategy has a significant impact on the implementation detail of the HP-UX layer. For example, MODCAL is used instead of C as the implementation language. However, user-level code written for System III UNIX will run on HP-UX, unless it depends on certain internal implementation details such as the directory format structure or invisible internal system data structures.

The advantages of this layering approach come in two main categories—leverage and opportunities for contribution. A large portion of hardware-dependent code was already written for the Series 500 and its peripherals. Using the SUN kernel made it unnecessary to rewrite this code for HP-UX. Existing modules used include device and interface drivers—especially significant because of the complexity of the HP-IB (IEEE 488) and the new HP CS-80 discs—low-level memory management, power-up code, process scheduler, architecturally dependent utility routines, and other machine-dependent code.

SUN has a number of features that are not present in UNIX; these features provide opportunities for HP-UX to make a contribution over other UNIX implementations. These include real-time performance in the area of interrupt response time and process switching, support for multiple CPUs, reliability in the face of system errors, support for variable-size independently managed dynamic memory segments, semaphores, and low-level device I/O capability. Also, HP's IMAGE data base management system was already implemented on top of SUN for the BASIC system. This code was ported to the HP-UX environment

What is UNIX™?

The popularity of the UNIX™ operating system developed by Bell Laboratories has been increasing since it became operational in 1971. Today, it is rapidly becoming the most popular operating system for mid-sized computers and runs on numerous machines made by different manufacturers. There have even been those that have likened UNIX's role in operating systems today to FORTRAN's role in computer languages some twenty years ago.

UNIX was developed by Ken Thompson and Dennis Ritchie of Bell Laboratories. Both men had been working on a project called Multics (an acronym for multiplexed information and computing service), which was a large multiuser operating system that was eventually cancelled by Bell Laboratories. From there, Thompson, and then Richie, went on to develop UNIX. As you might expect, many of the more desirable features found in Multics were incorporated in the UNIX design. In fact, even the UNIX name was adopted from a playful twisting of "Multics."

As the years went by, the UNIX systems within Bell Laboratories evolved until version six (V6) was developed about 1975. This version became quite popular in a number of universities around the world, including the University of California at Berkeley (UCB).

Version seven was released in 1978 and quickly replaced V6 in most installations. This version is the base for most of the commercial UNIX look-alikes, of which the Xenix system developed by Microsoft is probably the best known. It is also the version on which UCB built their popular enhanced versions of UNIX. Each UCB version released contained a few enhancements over the previous releases. UCB's versions are designated by xBSD, where x is the version number and BSD stands for Berkeley Software Distribution. 4.2BSD is the most recent.

In early 1982, Bell Laboratories released System III UNIX. This version is the base for HP-UX (HP's version of UNIX), although HP-UX also incorporates some of the nicer features found in UCB's 4.1BSD version.

System V UNIX was released by Bell Laboratories in 1983. Bell is guaranteeing that all of their future UNIX versions will be compatible with System V.

UNIX Popularity

Exactly why UNIX has become so popular is a hard question to answer, but the reasons probably include:

- Simplicity. The UNIX system can be broken into fairly small independent pieces. Each piece can be comprehended individually and at a pace that is comfortable for a user. Few users ever need to learn all the features provided by UNIX.
- Power. The pieces of the system can be connected synergistically and manipulated at execution time, the I/O can be redirected, the output of one process can be connected to the input of another (forming a "pipeline" of arbitrary length), processes can be executed in foreground or background, a command list can be developed and then executed when desired and as often as desired, etc.
- Flexibility. Pieces of the UNIX system are easily added, replaced, or deleted. System reconfiguration is quick and straightforward.
- Software. Bell Laboratories, Hewlett-Packard, and a lot of other companies and individuals have put a lot of effort into developing a large software base that runs in the UNIX environment.
- Ease of porting. Most of the UNIX system is written in a machine-independent manner. It has been ported to a number of different computer architectures with relatively few problems.

Features

UNIX has many features. Some of them are:

- The shell. The shell is a program that provides the interface between the user and the UNIX system. It is a command interpreter that takes input from the user and executes the requested commands. It can also take input from an ASCII command file, which is generally referred to as a "shell script." When a command is executed, it can be passed arguments, have its standard I/O files redirected, and/or be placed in the background, all through provisions built into the shell. The shell also has flow control structures that allow conditional and multiple execution of command lists. Because of the flexibility of UNIX, the shell can be replaced by a different program. In fact, UCB has chosen to do just that and provides their own version of the shell called the C shell.
- The C Language. C was developed concurrently with UNIX at Bell Laboratories. It is a medium-level language with many of the features found in Pascal and other high-level languages. It provides a programmer with a lot of power and few constraints. Most implementations of the UNIX kernel and most of the UNIX commands are written in C.
- Other languages. Currently HP-UX on HP's HP 9000 Series 500 and Series 200 Computers offers compilers for Pascal and FORTRAN 77 in addition to C.
- Full set of commands. Commands to maintain the UNIX system and the file system, editors, text processors, and numerous other commands are included in HP-UX. The popular vi editor from UCB is included in this set.
- A rich set of library routines. These include routines to compute common math functions, to perform formatted I/O, to access kernel intrinsics, and, on the HP 9000 Series 500 Computers, routines to manipulate virtual memory objects, to do DGL/AGP graphics, and to access an IMAGE data base.
- Data communication support. System III and other versions of UNIX provide a set of UNIX-to-UNIX copy (uucp) services to allow the user to pass files from node to node in a UNIX network. A sophisticated electronic mail system has been implemented by using these services. To these, the HP 9000 Series 500 Computers add a local area network (LAN 9000), general terminal emulator capabilities, and remote job entry.
- Source code control system (SCCS). This is a set of commands that helps the programmer keep track of changes to source files.

Further Reading

1. H. McGilton and R. Morgan, *Introducing the UNIX System*, McGraw-Hill, 1983. A good tutorial.
2. R. Thomas and J. Yates, *A User Guide to the UNIX System*, OSBORNE/McGraw-Hill, Berkeley, 1982. Another good tutorial.
3. *Bell System Technical Journal*, Vol. 57, no. 6, Part 2, July-August 1978. The entire issue is dedicated to UNIX of about version seven.
4. *HP-UX Reference Manual*, Hewlett-Packard Publication 09000-90004. A good reference, but not easy for a novice to understand.
5. *HP-UX Selected Articles*, Hewlett-Packard Publication 97089-90002. Nineteen articles on some of the large components found in UNIX.
6. S.R. Bourne, *The UNIX System*, Addison-Wesley, 1983. A good introduction.

-Michael L. Connor

to provide this important HP standard data base capability.

An important concern was the performance of a layered implementation; the risk was that conversion between the SUN format and the HP-UX format would increase operating system overhead. The experience actually observed after the product was completed was that the HP-UX layer itself is responsible for approximately 10% of the CPU time used by the kernel, and nearly all of that time is spent doing useful work such as loading programs. This means that SUN is a fairly good match for the HP-UX requirements, because little time is wasted on conversion between SUN and HP-UX formats.

Matching SUN and HP-UX

This section describes the areas of the SUN operating system that were changed or augmented to support the requirements of HP-UX. Only areas that are important to mapping the UNIX semantics onto the original SUN kernel are described in depth.

File System. There was already a good match between the SUN operating system and HP-UX in the hierarchical directory structure of the file system. The existing directory format was modified to fit HP-UX semantics rather than implement the standard UNIX disc format in MODCAL. The fundamental operations such as read, write, open, and close were already supported in a satisfactory manner in SUN; no significant changes to these were necessary.

However, the file system itself was the area that required the largest changes in SUN. One of the biggest additions was the support of device files, special files that map devices such as printers or terminals into the same name space as regular files. The SUN file system expected device and file accesses to be made separately. Special checks had to be made for special file types; the new device file code performs operations for device files equivalent to those originally performed only for regular files.

Another large change was support for mounting disc volumes onto an on-line directory so that all accessible files and directories are part of a single directory hierarchy. Again, special code was added to check each directory access; if the directory has another volume mounted on it, the access is redirected to the root directory of the mounted volume.

The third area of major change was file access protection semantics. The UNIX read/write/execute and user/group/other mechanisms used to control access to files were not originally in the SUN file system protection scheme. This could have been added, along with the standard UNIX disc format structure, to a separate directory format module, since SUN supports multiple directory format structures. However, the characteristics of the existing format were so close to those desired that the SUN format and protection scheme were adapted to the HP-UX requirements instead.

Changes were made in the SUN file system to support pipes and FIFO (first-in, first-out) files. In the early versions of HP-UX, pipes were implemented in the HP-UX layer. However, they have been moved inside the SUN file system for performance reasons. A number of minor HP-UX file system operations had to be added to SUN. These include changing the owner of a file, reading or changing file access modes, and duplicating an open file descriptor.

Some operations are performed in the HP-UX layer. These include parsing multilevel path names, managing the user's open files table, and enforcing file size limits on extending files.

I/O. In the area of device I/O, the existing SUN I/O system was a very good match for the needs of UNIX. Virtually no changes were made to the I/O primitives that provide the interface to the backplane and I/O processor, the bus bandwidth management code, the drivers for interface cards, or the disc and tape device drivers.

The major changes came in the internal and external terminal support. The external terminal driver is based on the existing serial interface driver, but adds UNIX tty semantics such as type-ahead, line buffering, mapping carriage return/line feed to newline, and sending the interrupt and quit signals. The Model 520 Computer's integrated keyboard and CRT device control code is based on the work done for the BASIC system's human interface. But the functional operation of the integrated "terminal" had to be completely redone to be compatible with HP terminals.

Memory Management. Because of the simple memory model of HP-UX, the memory allocation intrinsics are easily supported on most operating systems, including the SUN kernel. The major changes in the SUN memory management system were required by the addition of virtual memory and shared memory, which are extensions rather than semantic requirements of UNIX. The HP-UX layer has the responsibility of keeping track of the user's memory use and deallocating this memory when a process or program terminates.

Program Loading. No explicit function for loading and executing programs is present in the SUN operating system, but the underlying support needed is there. The file system is used (with minor changes) to find and read the program file, and the memory management system provides the mechanism for allocation of code and data segments. No major changes were required in the SUN kernel to support program loading.

The HP-UX layer manages shared code segments, which allow multiple processes to share a single copy of the code. The HP-UX layer also handles relocation of code and data segments at load time and meets the segment attribute requirements requested by the object file format.

Process Management. The HP-UX process management intrinsics are supported fairly well by the SUN kernel, but two areas required a significant effort: fork and signal. The fork system call creates a new process in the exact image of the calling process. It returns to both the parent and child processes, just after the fork call, at the point where the function return value distinguishes the child from the parent. Creating an exact copy of a process is not a typical operation supported by normal operating systems, including the SUN kernel.

At the SUN level, code was added to support the "cloning" of a process. The cloning operation allocates memory for the child process and initializes SUN modules for the new process. It is also responsible for duplicating the contents of the parent's segment table in the child's segment table and creating an exact image of all the parent's segments in the child's address space, including virtual memory segments and the stack segment.

The HP-UX layer then initializes the new process. This includes allocating an HP-UX process control block, copying some fields from the parent's process control block, and initializing other unique fields such as process ID and parent process ID. It also increments use counts on shared objects such as shared code segments and open files. Finally, the HP-UX layer returns the appropriate value to the parent (child's process ID) and to the child (zero).

Signal Implementation. The implementation of signal, a mechanism for interprocess event notification and exception reporting, was a significant portion of the HP-UX layer development. SUN had no explicit support for sending asynchronous signals between processes, but did have most of the tools necessary to implement this feature.

One tool is the ability of subsystems to install trap handlers for most classes of traps possible on the Series 500 Computers. Signal processing is initiated by triggering an MI (machine instruction) trap in the target process, which causes the MI trap handler to be entered on the next machine instruction executed. This handler is responsible for processing the signal received and taking the specified action. This can be calling a user-specified signal handler, terminating the process, or just ignoring the signal.

Other Process Management. The process scheduler met the requirements of HP-UX in the original SUN implementation, but has been improved to allow dynamic process priority adjustment to reward interactive processes. (It is currently being enhanced to suspend low-priority processes during heavy system loads.) SUN supports the creation of special system processes that can provide specific system services. These system processes communicate with user processes and each other via SUN's mailbox-style interprocess messages. Also, a sophisticated set of semaphore operations is provided for synchronization of all processes in the system. This is especially important in a multiple-CPU system; merely disabling interrupts does not ensure exclusive access to a shared data structure, because other processes may be running simultaneously on other CPUs.

The following process management functional areas are implemented in the HP-UX layer:

- Higher-level support of fork such as allocation and initialization of a process control block for the new HP-UX process
- Higher-level support of signal, including sending and receiving signals, and specifying action to be taken on receipt of a signal
- Management of user, process, and group IDs
- Process termination, including deallocation of resources owned by the user process
- Wait for a signal or for termination of a child process
- Management of HP-UX process control blocks.

The functional areas listed below are completely supported by the SUN kernel, except for those changes noted.

- Power-up
- Multiple-CPU support
- Trap handling
- Real-time clock: the HP-UX layer performs the conversion between SUN time format and HP-UX time format
- Alarm clock: the HP-UX layer creates a system process that wakes up each second to see if any alarm signals need to be sent
- CPU times; a minor change was made to the timer interrupt service routine to increment the CPU time used by the current process.

Upper-Level Software Strategy

Working in parallel with the SUN and HP-UX kernel design groups was another group of software engineers who were responsible for the upper-level commands and libraries. The UNIX system from Bell Laboratories contains more than 300 commands and over 200 library subroutines. Consisting of more than 300,000 lines of C source lines, these constitute the bulk of the UNIX system. The majority of HP-UX upper-level software on the Series 500 Computers is based on these UNIX System III commands, plus several from the 4.1BSD version of UNIX from the University of California at Berkeley (UCB).

For implementation priorities, the upper-level software team first categorized the commands and libraries into different groups based on their usefulness. For example, initialization and file manipulation commands were all in the first group. Useful tools were in the second group and other commands and libraries, such as those used for text processing, were in the third group. Then the C source code of the first two groups was studied in some detail using a C cross referencer to determine which system intrinsics and libraries were used. The data resulting from the study was stored in an HP 9845 IMAGE data base from which many useful reports were produced. For example, a system intrinsic implementation priority list was generated based on the highest-priority commands to guide the kernel group in their implementation. As new system intrinsics were brought up, the upper-level software team was able to determine from the data base what additional commands could be brought up with the newly available intrinsics.

Another IMAGE data base was used to keep track of all commands and libraries in terms of implementation priority, responsible engineer, porting status, source origin, etc. This proved to be very useful for managing the project and keeping other departments informed about the status of each command.

Porting Commands and Libraries. Four major tools were necessary to port the upper-level software: a C-to-HP-9000 cross compiler, an assembler, a linker, and a cross compilation machine. The upper-level software team used a remotely accessible VAX/750 running UCB UNIX as the cross compiling environment. Other tools to move files to and from the VAX/750 were developed as necessary.

After the initial system was up and running, the major focus was to make the C compiler resident on the Series 500 by cross compiling it. We had a resident environment two months later. From that point on, all development work was done on a Model 520 Computer running the latest (sometimes experimental) kernel. The upper-level software development system then grew from one single-user system to two multiuser systems linked with a local area network.

The majority of the commands and libraries were ported over to the Series 500 with little or no modification, that is, most of them ran after compilation. However, the following types of changes were necessary.

HP-UX: A Corporate Strategy

With the introduction of HP-UX on the HP 9000 Series 500 Computers, Hewlett-Packard has made a strong commitment to the use of an enhanced version of UNIX™ as a standard operating system for its new computer products. Through this commitment, HP is striving to eliminate unique software attributes that make end-user programs difficult to "port" from one computer to another. Programmers can now design their software to run on an array of HP machines, concentrating on modularizing and scaling their applications to best suit each computer's price/performance characteristics.

Why UNIX?

Since any operating system standard would simplify the porting process and improve programmer productivity, why was UNIX selected as the heart of HP's software strategy?

UNIX is gaining wide acceptance as an industry standard for 16-bit and 32-bit minicomputers. Its popularity is partially because it has been easy to implement on a variety of processors and computer architectures. This portable characteristic made UNIX an ideal choice as a compatible operating system for the distinct architectures of current HP 9000 members: the 16/32-bit 68000 microprocessor-based Series 200 Computers (Models 220 and 236) and HP's proprietary 32-bit VLSI-based Series 500 Computers (Models 520, 530, and 540). UNIX is also planned for future members of the HP 9000 family.

The popularity enjoyed by UNIX has a synergistic effect. Software applications are being designed for the UNIX environment at an increasing rate, which in turn encourages more UNIX implementations. Most of this software will run on HP-UX, thereby making HP's computers more attractive to a larger audience. Furthermore, UNIX is studied and taught in most major universities. Today's computer science graduates will eventually influence or become those who select computers for commercial and scientific use. UNIX-based products are likely to receive strong consideration during the selection process.

What Is HP-UX?

HP-UX is a combination of Bell Laboratories' UNIX operating system, portions of the University of California at Berkeley (UCB) implementation of UNIX and Hewlett-Packard software enhancements. Through UNIX, HP-UX facilitates easy importation of UNIX-derived programs and offers a consistent, powerful program development environment. Complementary extensions address the Manufacturer's Productivity Network (MPN), HP's view of how computer systems can be used in manufacturing organizations to improve productivity.

Rather than implementing every function of Bell Laboratories' System III UNIX, features were included based on their importance in porting standard software or their absolute program development value. Using these guidelines, a compatibility hierarchy was developed in which kernel services became a "must," library subroutines a "high want," and commands a "want."

As a result of this approach, HP-UX includes all System III kernel intrinsics and all libraries except for a handful of graphics subroutines. More than 125 of the most useful System III commands and a small but important number of UCB commands are also offered.

To satisfy customer requirements, enhancements covering programming languages, graphics, data base management, device and instrumentation I/O, local area networking, and friendly user interfacing are being standardized. These extensions, which appear as additional kernel intrinsics, libraries, and commands, will bridge the gap between HP's HP-UX and non-HP-UX computers.

Additional enhancements assist in migrating applications software from current proprietary HP operating systems to HP-UX. One of these tools, the Applications Migration Package (AMP), converts the HP 1000 Computer's RTE calls to HP-UX calls. AMP revisions are pianned as HP-UX is expanded to meet real-time control requirements.

New software features are not the only form of HP enhancements. On-going training allows sales and technical support organizations to provide complete services before and after sales. Easy-to-read tutorials and reference manuals aid both novice and experienced users. Exhaustive R&D software testing ensures reliable operation and minimal downtime.

Since HP-UX is planned for many future HP computers, HP will leverage investments already made in these important support areas. By avoiding the massive reinvestments continuously

UNIX is a U.S. trademark of Bell Laboratories.

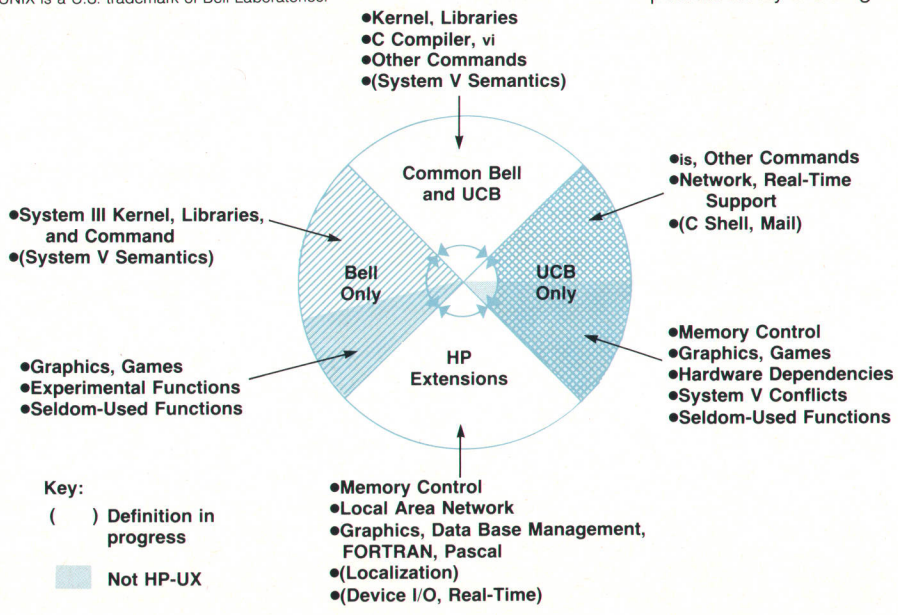


Fig. 1. *Influence of Bell Laboratories, UCB, and HP extensions on the direction of the HP-UX definition.*

required of new software systems, HP can concentrate on improving all aspects of HP-UX in the future.

HP-UX Standards Enforcement

Compliance with the HP-UX standard is enforced through comprehensive sets of validation programs. Automated test programs monitor proper operation of all kernel intrinsics, System III libraries, two-dimensional and three-dimensional graphics libraries, and the FORTRAN and Pascal compilers. As the standard evolves, additional validation programs will be developed to ensure consistency across all HP-UX computers.

Overall management of the standard is the ongoing responsibility of the HP-UX Steering Committee. Consisting of representatives from several HP divisions, this committee meets monthly to resolve pertinent HP-UX issues and to review the status of the various HP-UX working groups. These groups, also with broad divisional representation, cover technical, marketing, documentation, and customer support issues in more detail. Each division works through its representatives to propose additions or changes to the standard.

Future Direction

Perhaps the most critical issue in establishing the future course for HP-UX is its degree of compatibility with Bell Laboratories and UCB. While 4.2BSD UNIX (Revision 4.2 Berkeley Software Distribution) is currently the superior version, Bell is developing improved versions that could eventually surpass 4.2BSD in capability and reliability. In addition, four microprocessor manufacturers* intend to offer System V, Bell's latest UNIX version, on their microprocessor products. System V can potentially become the most affordable UNIX and thus the UNIX of choice for portable application programs.

*Intel, Motorola, National Semiconductor, and Zilog.

In consideration of these factors, the Bell System III version has been chosen as the base standard. The compatibility hierarchy will determine which portions of System V and its successors are HP-UX candidates.

Extensions beyond the Bell versions can be expected if they fail to meet HP requirements in a timely fashion. However, we prefer to adopt an existing UNIX-based implementation (if one exists) before embarking an an original design project. A potentially rich source of enhancements currently under investigation is UCB's 4.2BSD version. We anticipate adding such UCB features as the C shell, mailer, and selected kernel intrinsics.

Microsoft's Xenix, with its large installed base and potentially rich source of UNIX applications programs, could influence the HP-UX standard. Since Xenix and HP-UX are selectively adding Bell System V and UCB features to the same System III definition, conformance between the two systems is likely.

Fig. 1 illustrates the major influence of the HP extensions and the Bell releases on the HP-UX direction. It also recognizes UCB as a promising contributor of additional functionality.

In support of low-cost computer systems, we are examining methods of subsetting HP-UX without sacrificing compatibility or easy growth to the higher-performance systems. Code compaction and reduction techniques for both the operating system kernel and the disc resident commands are being considered. An exciting technique under investigation is a high-performance distributed HP-UX operating system, which allows individual workstations to rely totally on shared network peripherals. Thus, the cost per system is dramatically reduced, but local processing power is maintained.

HP-UX will be modified to support several European languages and the 16-bit Kanji character set. Thus, localized application program solutions will be possible.

-Michael V. Hetrick

- A new system intrinsic entry point mechanism was developed because the kernel was written in MODCAL and the rest of the system was in C.
- Some data structures contained in the C header files needed to be modified to match the HP-UX layer data structures. (Header files contain data and structure declaration statements for C programs.) The commands that needed these header files were examined in detail to see if modification was necessary.
- A few commands were rewritten completely because the kernel was not the original standard kernel. For example, fsck, the file system integrity checker and maintainer, was rewritten because the SDF (structured directory format) file system is physically different from the UNIX file system. The process status command ps was modified extensively because of data structure differences. Another example was the mknod command which creates special files to communicate with I/O devices. It was modified to match the UNIX semantics to HP-IB I/O devices. However, all the commands were kept as compatible as possible with System III UNIX commands.
- The Series 500 supports IEEE floating-point format; as a result, the UNIX math library was replaced with HP's own implementation.
- Twenty-one new commands were implemented that apply to the Series 500-based HP-UX. These deal primarily with machine-dependent features such as disc boot area management, disc initialization, setting virtual memory parameters, and system installation and update. The handling of DC600 tape cartridge data on HP's new CS-80 discs also required special support.

Problems During Porting. The problems encountered in porting the commands and libraries can be categorized in two areas—architecturally dependent and architecturally independent. Architecturally independent problems were mostly anomalies found in the original UNIX code. We logged over 281 new bug reports during the port project. Over 60% of these bugs were fixed. The others were either classified as not worth fixing or waiting to be fixed.

Architecturally dependent problems were usually caused by dereferencing of nil pointers or dependency on the direction of stack growth. On the VAX/750 implementation of UNIX, a nil pointer dereference returns a zero. On the HP 9000 Series 500 HP-UX, a system trap occurs. This architectural dependency is relied on in many places in the standard UNIX commands and libraries, and each of these needed to be corrected. These usually manifested themselves in a memory fault error message. Fortunately, this error was relatively easy to fix in the source code.

The stack grows towards high memory (up) on the Series 500 and down on the VAX/750. For example, the printf subroutine in the standard I/O library can have a variable number of parameters and the pointer used to access the parameters on the stack is decremented rather than incremented. Other architecturally dependent features included the byte order swap of the VAX/750 hardware where

low and high bytes are reversed. This made reading cpio archive format tapes from the VAX/750 a chore in the beginning. Now HP-UX defines a new -p option to the cpio command which does the byte swap.

The upper-level software team did not have a user-level debugger available to debug the C programs. Instead, the kernel-level HP 9000 debugger was used to debug the commands. It was cumbersome to set up the initial breakpoint, but quite effective after that. (A user-level symbolic debugger is being developed.)

Shared Libraries. The Series 500 architecture supports shared code segments, thus allowing the implementation of a special shared library for major portions of the standard C library. That is, there is only one copy of the library in the system shared by all system commands that are linked in the standard C library. (The shared library feature is not currently available to user programs.) This saved typically 7K bytes of code space for each command (just about all of the commands used the C library). This, in turn, improved load-time performance and saved disc space.

SCCS and the Build Process. UNIX is touted as one of the best program development environments available, because it provides many software engineering tools. The source code control system (SCCS) is one such tool that the upper-level software team took advantage of throughout the project life cycle. The SCCS was brought up and used as soon as all kernel support was available. The Bell Laboratories System III source code was put under SCCS as the baseline and all upper-level software changes were built on top of it. Each upper-level software team member adhered to a simple set of rules that applied to the access and update of the controlled source. This proved valuable for day-to-day software development, providing who, when, how, and why information about code changes.

SCCS maintains revision numbers to allow access control and retrieval of any version of the source code. It also supports checksums of the source files to check for corruption. This was important since code development was done in parallel with the file system development and the checksum is a simple physical integrity check. SCCS was indispensable later during quality assurance testing and the code freeze period just before each major system release.

System build scripts were written to manage the compilation of all the commands and libraries from the SCCS source directory automatically. The build procedure, along with the scripts, was able to handle compiler, assembler and linker updates, getting the source, and compiling the system in proper sequence. This was important for system-wide changes such as object file format changes or major updates in the compiler or other tools. The scripts also controlled the target file system structure, setting file ownerships, access permissions, etc. They also managed the SCCS update revision level of each system build such that any change occurring after the build started would be at a higher level and would not be included in the current build even if the build process had to be restarted for some reason. The build scripts evolved through the life of the project and became a major tool for system releases. The final build of the 3.3M-byte system took around 17 unattended hours to complete.

Compatibility

The upper-level software porting experience indicated a high degree of compatibility between the HP-UX layered kernel and the UNIX System III kernel. Out of 126 ported commands from System III, 57 required no modification at all, 44 required less than 10 lines of modifications, 16 required between 10 and 30 lines of modifications, and 9 required more than 30 lines of modifications. Most modifications were to fix bugs. These commands do not include development tools such as a compiler, an assembler, and a linker, nor do they include UCB UNIX commands.

Extensive effort was made to ensure compatibility with Bell Laboratories' System III UNIX. First, a "minimum touch" strategy on the System III source code was used. The design team did whatever was necessary to make the commands and libraries work, but beyond that they did as little modification as possible. Temptations to clean up the code were strongly discouraged. Each reported bug was evaluated to determine whether it should be fixed and if so, how.

Second, validation suites† were used to ensure compatibility with System III. The priority for the validation suites was to validate the kernel first, then the libraries, and finally the commands. 100% of the kernel intrinsics were validated. A significant effort was invested in the kernel validation suite. It was run after each new kernel was built. 92% of the subroutine libraries have validation tests and all are incorporated into an automatic test suite. 22% of released commands have validation tests. The validation suites were written with verification of the functionality in mind rather than exhaustive quality assurance testing.

The automatic validation test suite is organized for ease of use. There are two types of tests—one related to the root user* and the other related to the typical user. The automatic test suites were provided to the software system integration team for testing commands and libraries with other major subsystems.

Acknowledgments

We would like to thank the following people for their contributions to the HP-UX effort. Fred Clegg at HPDA (HP Design Aids) spearheaded the UNIX effort in HP. Rich Hammons and Richard Tuck, then at HPDA, did the early work on the C cross compiler, assembler, and linker. Bill Williams at HPDA worked on the kernel validation suite. Everyone associated with the upper-level software effort worked on commands one way or another: Debbie Bartlett worked on the libraries and build scripts and managed the system build process, Xuan Bui worked on the command and automatic validation suites, Mike Connor was the lead engineer and did the initial work on the MODCAL and C compilers and assemblers, the shell, and the fsck subroutine, Kathy Harris worked on the SCCS and then did the subsequent work on the C compiler and the assembler, John Harwell was the system manager and worked on the tcio and command validation suites, Ken Lewis did the system performance characterization, J.L. Marsh worked on the vi editor, systems integration, and installation tools, Mike McCarthy worked on the MODCAL and HP-UX linkers, Marty Osecky did the subsequent work on the assem-

†A suite here refers to a set of programs for a common purpose.

*Referred to as a super user in UNIX, a user exempt from normal security checks.

bler, Don Rosenbaum and Rick Dow from HP's Colorado Networks Operation (CNO) worked on RJE and the uucp and cu datacomm commands, Alan Silverstein did the work on the HP 9000 debugger, system manager, tool builder, and systems integration, Ron Tolley worked on system performance characterization, and Helen Vu worked on the schema and library validation suites.

The following people contibuted to the HP-UX kernel effort: Mike Berry, HP-UX file system, Barb Flahive, pipes, magnetic tape and printer support, Milan Hanson, bug reporting data base, Mark Hodapp, extended memory intrinsics, Karl Jensen, program loading, Ken Martin, IMAGE data base management system, and Peter Notess, terminal and integrated CRT and keyboard drivers.

We would also like to thank Jack Cooley for managing the C compiler and system performance characterization efforts. Dave Graham from HP's Data Systems Division (DSD) managed the FORTRAN and Pascal compiler efforts, Ken Heun managed the initial HP-UX kernel effort, Jim Willits and Vince Jones from CNO managed the local area network and datacomm efforts, Mike Kolesar managed the systems integration and release process, Eileen McGinnis from DSD managed the AGP/DGL graphics effort, Denny Georg and Dan Osecky managed the SUN kernel VM effort, and Mike Hetrick managed the entire HP-UX program.

An Interactive Run-Time Compiler for Enhanced BASIC Language Performance

by David M. Landers, Timothy W. Tillson, Jack D. Cooley, and Richard R. Rupp

AT THE BEGINNING of the BASIC project for the HP 9000 Model 520 Computer, the project team was faced with a major challenge. To take full advantage of the performance available in the Model 520 from the new 32-bit NMOS-III VLSI microprocessor,[1] BASIC had to be implemented as a compiled language. Using traditional compiler technology, this would mean giving up many of the interactive features so popular with current HP 9845 users. The challenge was to develop a new compiler technology that would support these interactive features while maintaining the performance advantage of a compiler.

The breakthrough came in the form of two articles on "throwaway compiling," explained in two articles—one by P.J. Brown[2] and one by J. Hammond.[3] The throwaway or run-time compiling technique compiles each line the first time it is executed. As more of the program is compiled, the performance approaches that of a traditional compiled system. If the program runs out of memory, the current object code is discarded (hence the term "throwaway compiling") and the incremental compilation is restarted at the next line to be executed.

The authors were looking for a way to run programs efficiently on machines with limited memory space, but the throwaway compiling technique looked like it could be adapted for a run-time compiler that would provide the desired interactive features. If the object code could be thrown away during the execution of a program and rebuilt without restarting, it could also be thrown away at arbitrary times such as when the user modifies the program. Within limits, the program reconstructed after throwaway could be different from the program before throwaway. This would support the pause, edit, and continue feature. Given that an intermediate form of the program is available to reconstruct the object code at run time, this intermediate code could be designed to contain enough information to support the interactive debugging features. Finally, if the object code could be constructed one line at a time and added to the object code at run time, the code for a single line could be constructed and immediately executed as well. This would allow asynchronous execution of single lines from the keyboard during program execution.

Enhanced BASIC Language

The BASIC language that Brown implemented as part of his research was a very minimal subset, whereas Model 520 BASIC is a substantial language with several significant features beyond those supported by most other BASIC systems. Could these more advanced features be implemented in a run-time compiling environment? That was the ultimate challenge facing the design team. Some of the language features that presented the biggest challenge were:

- Subprograms similar to FORTRAN routines, but supporting recursion. Both subroutine and function subprograms are supported.
- A COMMON statement similar to that used in FORTRAN. Both blank and labeled COMMON are supported. EQUIVALENCE is not supported.
- ON conditions; a mechanism for handling asynchronous interrupts within a BASIC program. The interrupt service routines are part of the program, accessible via GOTO, GOSUB or CALL statements. Normal program flow can be altered at any line boundary in response to one of these

interrupts. Examples of possible interrupts include keyboard keystrokes, interrupts from I/O devices, software signals, and real-time clock events.

- Structured programming constructs such as IF/THEN/ELSE, WHILE and REPEAT loops, and CASE.
- A REDIM statement that can dynamically change array bounds.
- Dynamic variable allocation/deallocation via the ALLOCATE and DEALLOCATE statements.

User Code Structure

The internal representation and management of the user's program in the Model 520 BASIC system provides insight into a complex and fascinating software architecture. This representation is called the program chain, which is a collection of contexts,* each of which represents a user-level subprogram. A context can either be compiled, or in a form from which the original source code can be reconstructed, called intermediate code (icode). Compiled contexts are created using the COMPILE command (not to be confused with the code compiled by the incremental run-time compiler), and are discussed in greater detail below. The icode contexts can be listed and modified at the source level by the user; the name comes from the fact that the source is represented internally in a form that is midway between source and object code. The icode contexts also contain the incrementally compiled object code produced by the run-time compiler as the program runs.

Intermediate Code Contexts. An intermediate code context consists of two machine data segments: the icode segment and the symbol table segment (see Fig. 1).

The context header holds information that describes the context and its relationship to the other contexts in the program. Also in the context header is a pointer to the corresponding symbol table segment and to the next and previous contexts in the program chain. The static object code contains many small code sequences needed to support running BASIC programs, including code to handle ON conditions, end the program, handle input responses, and other tasks. This static object code is always there, and the incrementally compiled object code branches to it when in need of some help for one of these tasks.

*The reader should be aware that other articles in this issue may define the term "context" differently.

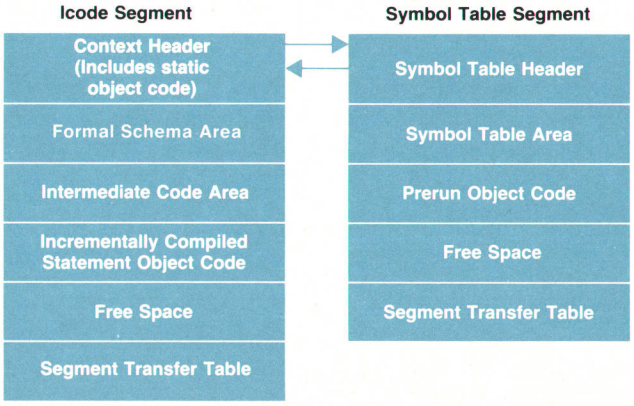


Fig. 1. *The icode context contains two segments as shown.*

The formal schema area holds a compact description of the parameter list for this subprogram. It describes the number and types of the parameters and is useful for supporting the call linkages. The icode area holds the representation of the lines of the user's subprogram. Each line of source corresponds to one line of icode. Whenever the user modifies the intermediate code, the object code gets thrown away. The intermediate code can then grow or shrink without having to move the object code. The incrementally compiled statement object code is the object code for the statements in the context. As the program runs, the object code builds up in this area. The segment is extended if necessary to make room for more object code.

The free space contains all the unused space in each segment; all the other areas are directly adjacent. The object code for a keyboard command goes into this area. Since a command is a one-time event, and not part of the program, the object code for that command disappears after the command is executed. If there is not enough empty space to hold the command's object code, the segment is increased to make room.

The segment transfer table holds the pointers to procedures for calls into and out of a segment. During incremental run-time compilation, this table grows and may cause a segment extension.

The symbol table header contains a pointer to the icode segment, the total size of the symbol table segment, and lengths of items in the symbol table. The symbol table area contains a series of entries, one for each identifier in the context. There are fields in the entry for the storage organization of the identifier (e.g., COMMON and ALLOCATED), the identifier representation such as DOUBLE or REAL, the number of dimensions (if an array), the type of identifier (label, numeric variable, subprogram, etc.), the offset into the value area of its definition, and the characters of the identifier name. If this area has to grow because the user enters new identifiers, it moves the prerun object code down, extending the segment if necessary.

The prerun object code allocates space for the local variables of the context, and it also initializes any bounds that these variables need. This object code does not correspond to any program statement; it just sets up the variables that the statement object code will use. In BASIC, variables do not have to be declared explicitly; new variables can be defined by keyboard operations or even by modifying an executing program. This run-time implicit variable allocation can cause the prerun object code to grow so that the new variables can be initialized at the next activation of this context.

The double segment approach facilitates the management of all the dynamic edges. All areas except for the two headers must be able to grow. The icode area and the symbol table area must grow at the same time during parsing. The statement object code area and the prerun object code area must grow simultaneously during program execution.

Compiled Code Contexts. The user can compile any context currently in memory by using the COMPILE command and store the object code in a PROG format file. Two benefits accrue from the fact that compiled contexts contain no intermediate code. They require less memory when loaded, and it becomes possible to release programs without releas-

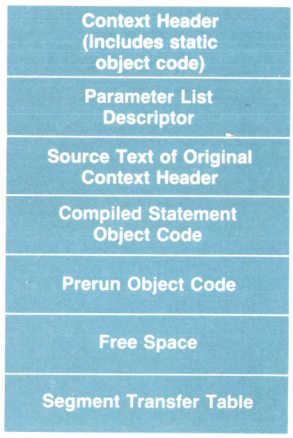


Fig. 2. *Machine data segment for compiled code context.*

ing their source. Compiled and icode contexts can coexist in the same program. In this case the icode subprograms list normally, while the compiled ones list the source of their original context header. These lines begin with >>>>> to indicate that the subprogram code is compiled.

Since compiled contexts have fewer dynamic edges than their intermediate code counterparts, they require only one machine data segment (see Fig. 2).

Icode Format. Each context contains a block of intermediate code that directly represents the source text of the original subprogram. There is one line of icode for each line in the source. A line of icode contains a header, followed by a series of tokens that represent keywords, operators, constants, and symbol table entries. These tokens are of varying length and are generally in the same order as the elements they represent in the original source, except for expressions, which are in reverse Polish notation (RPN). The first byte of each icode token describes what type of entry it is and how many bytes the entry takes.

The combination of RPN for expressions and source order for everything else in the intermediate code may seem strange. Since the Model 520's CPU uses a stack architecture, RPN makes it easy for the compiler to generate optimal code for expressions. On the other hand, source order simplifies listing and nonexpression code generation, because the compiler can know what kind of statement it is dealing with at the beginning of the icode line.

A line of icode is simply a series of bytes from 11 to 255 bytes long. There are length fields in each line to allow the system to traverse the lines of icode either forwards or backwards. This last capability is useful when scrolling backwards in the editor. The system generally refers to a line of icode by specifying its offset in the icode area.

The objective in the design of the intermediate code was to minimize the memory space it requires. Most program elements need just a single-byte entry to represent them. For numeric constants, studies have shown that most constants are small integers. Thus, for integer constants in the range 0 to 9, single-byte icode entries are used. For the somewhat larger constants (up to 255), two-byte entries are used. Constants greater than 255 require five-byte entries. Floating-point constants are represented as character strings. Most real constants such as 5.3 only have a few characters, so storing them as characters takes fewer bytes of storage than if they were stored as an eight-byte real value. Keywords are arranged so that the most common ones have a single-byte icode representation. All other entries take either two or three bytes.

Symbol table entires have two possible forms. In BASIC programs, commonly referenced identifiers tend to have single-letter names such as I, J, and N, and represent numeric variables. Ten special locations are reserved in the symbol table for this type of identifier, and a special single-byte icode entry exists to represent them. All other identifiers need a two-byte icode entry. If there are more than ten single-character numeric variables, the first ten will use the single-byte representation, and the rest will use the two-byte representation. All nonnumeric identifiers, such as strings, labels, functions, and subprograms always use a two-byte icode representation.

Two examples of icode program lines for two typical BASIC statements are shown in Fig. 3.

Fundamental Mechanisms

The run-time compiler is an incremental compiler. That is, the program is compiled one piece at a time. In this case the unit of compilation is a BASIC program line and each line is compiled the first time it is executed. The simple program listed in Fig. 4a illustrates the fundamental mechanisms of the run-time compiler. As a programmer enters a program, it is translated from the BASIC source code to an intermediate code representation as discussed earlier. When the programmer presses the **RUN** key to execute the program, the system detects that the first line of the program is not yet compiled, so a bootstrap code sequence is emitted to invoke the compiler to compile the first line (Fig. 4b) and control is passed to it. Line 10 is then compiled and the compiler checks to see if the next line, line 20, has been compiled yet. It has not, so a code sequence to invoke the compiler for line 20 is appended to the end of the code for line 10.

This new code overlays the initial bootstrap sequence, which is no longer needed (Fig. 4c), and control is transferred to the code for line 10, which is executed and then

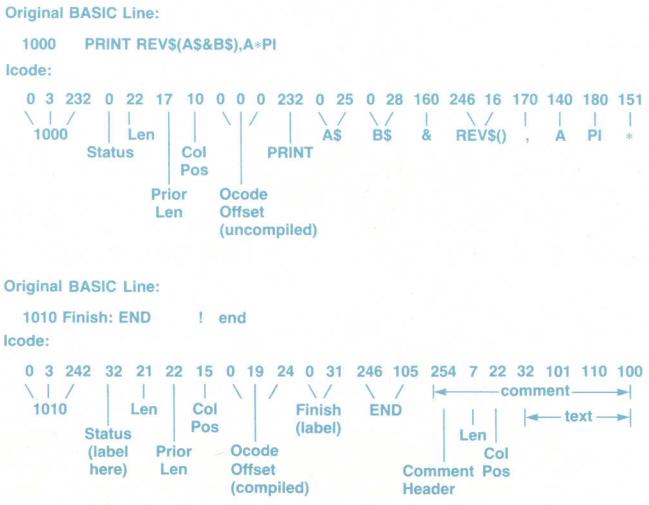


Fig. 3. *Two examples of icode representations of BASIC program lines.*

(a) Example BASIC program

```
10 PRINT "Table of Squares and Square Roots"
20 I = 1
30 IF I > 100 THEN Done
40 PRINT I,I^2,SQR(I)
50 I = I+1
60 GOTO 30
70 Done:END
```

(b) RUN command bootstraps to begin execution

```
call compiler(10)
```

(c) Line 10 compiled

```
code for line 10
call compiler(20)
```

(d) Line 10 executed and line 20 compiled

```
code for line 10
code for line 20
call compiler(30)
```

(e) Line 20 executed and line 30 compiled

```
code for line 10
code for line 20
code for line 30
  test for I > 100
    true: call compiler(70)
    false: call compiler(40)
```

(f) Line 30 executed (test was false) and line 40 compiled

```
code for line 10
code for line 20
code for line 30
  test for I > 100
    true: call compiler(70)
    false: code for line 40
call compiler(50)
```

(g) Line 40 executed and line 50 compiled

```
code for line 10
code for line 20
code for line 30
  test for I > 100
    true: call compiler(70)
    false: code for line 40
code for line 50
call compiler(60)
```

(h) Line 50 executed and line 60 compiled

```
code for line 10
code for line 20
code for line 30
  test for I > 100
    true: call compiler(70)
    false: code for line 40
code for line 50
  branch to code for line 30
```

(i) Line 30 executed (test was true) and line 70 compiled

```
code for line 10
code for line 20
code for line 30
  test for I > 100
    true: branch to code for line 70
    false: code for line 40
code for line 50
branch to code for line 30
code for line 70
end program
```

Fig. 4. *Example of run-time compiling. See text for explanation.*

follows through to invoke the compiler for line 20. Similarly, line 20 is compiled (Fig. 4d) and executed and the compiler is called to compile line 30. After line 30 is executed, there are two different lines that may be executed next, depending on the results of the IF test. Therefore, the compiler emits code to invoke itself for both lines 40 and 70, and the IF test will branch to one piece of code or the other (Fig. 4e). Because the initial value of I is 1, the test is false the first time line 30 is executed, so the compiler is called to compile line 40. Lines 40 and 50 are compiled and executed (Figs. 4f and 4g) and the compiler is then invoked to compile line 60.

Line 60 is an unconditional transfer of control to line 30, which the compiler realizes is already compiled. Therefore, a branch instruction to the code for line 30 is all that is emitted for line 60 (Fig. 4h). The main loop in the program is now entirely compiled, so the next 99 times through the loop execute only compiled code, allowing the performance of the system to be essentially the same as the performance of a traditional compiled system.

Once the value of I reaches 101, the test in line 30 is true, causing the compiler to be invoked to compile line 70. In this case, the code for line 70 cannot directly overlay the call to the compiler, because doing so would overlay code for other program lines. Instead, the code of line 70 is appended to the end of the rest of the compiled code and the call to the compiler for line 70 is replaced with a branch instruction to the code for line 70 (Fig. 4i). The program then terminates, but the compiled code is still present. If the user chooses to rerun the program, the RUN command now finds that the first line is already compiled and transfers control directly to it so that the second execution of the program executes only compiled code.

Interactive Features

In traditional interpretive systems, special checks for user interactions or tracing take place at the beginning of each line. Checking one or more flags can be done with just a few machine instructions, which require a very small overhead compared to the overall execution of the interpreter. In a compiled environment even a few instructions can consume a large percentage of the total execution time of the program. The solution developed in cooperation with the CPU microcode team was the start-of-line-check instruction SOLC. This instruction is the first instruction of every compiled BASIC line. It performs two important tasks. One, it checks a word at the base of the stack for zero versus nonzero. If one or more bits in the word have been set, indicating that something special needs to occur, a trap occurs and the system takes the appropriate action. If the word is zero, execution proceeds to the next instruction. Second, the SOLC instruction writes its own address at a fixed location in the stack so that the system can always find out which line is being executed.

A traditional interactive feature on HP desktop computers has been the live keyboard. The user can evaluate expressions, examine and modify program variables, and execute BASIC statements from the keyboard while the program is running. When a Model 520 user presses the **EXECUTE** key after typing in a command, one of the bits in the SOLC check word is set, causing a trap to occur at the next

SOLC instruction. The system then parses, compiles, and executes the interactive command before returning control to the program line that was interrupted.

Another traditional HP interactive debugging feature is the ability to trace program flow by enabling the TRACE mode. This causes a message to be displayed on each nonsequential transfer of control, showing the source and destination line numbers. When a Model 520 programmer enables tracing, another bit in the SOLC check word is set which causes a trap on every SOLC instruction. The system can then determine whether or not the BASIC line currently being executed is immediately after the previously executed line, and display an appropriate message if it is not.

Another important debugging capability is the ability to trace the assignments to program variables. When the programmer enables variable tracing, the system enters a mode where a trap occurs on every store into a memory location. The system can then determine if the location is the location of a program variable, and if so, display a message with the new value of the variable and the line number of the line that changed the variable.

Although enabling either or both of the tracing modes slows down program execution speed significantly, the program usually executes faster than the programmer can follow it unless the trace messages are slowed down with the TRACE WAIT statement, which causes a delay after every trace message is displayed.

The occurrence of an asynchronous ON condition also causes a bit to be set in the SOLC check word. When the next SOLC instruction executes, a trap occurs and the system sets up a branch to the specified service routine if the scope and priority conditions are satisfied. The system transfers control to a piece of static object code at the beginning of a context, which in turn branches to the service routine if it is already compiled, or to a bootstrap sequence to invoke the compiler if it is not yet compiled. CALL or GOSUB branches invoked by the ON condition return to the point of interruption as directed by the static object code after handling the ON condition.

Program Modification and Continuation

While debugging a program, a programmer often wants to be able to make a fix to the program and resume execution without having to start the program over. The run-time compiler allows the Model 520 Computer to support this capability with a compiled system. As an example, suppose the author of the program in Fig. 4a decided during the execution of the program to calculate the squares and square roots for all integers up to 1000 instead of 100 as in the original program. Suppose that the program was at line 40 when the programmer entered the editor and changed line 30 (Fig. 5a). The compiled code for the program is no longer valid, so it is thrown away. The system remembers that the program is currently at line 40. When the programmer continues the program, the system determines that line 40 is not compiled and sets up a bootstrap sequence for line 40 similar to the way in which the program first began execution with line 10 (Fig. 5b). Line 40 is recompiled and executed, followed by line 50 and so forth (Figs. 5c to 5g). The compiled code is rebuilt a line at a time, just as it was constructed the first time.

There are some restrictions on what lines can be changed while a program is running. Lines that have only partially been executed cannot be modified or deleted. For example, a line that invokes a multiline function or a subprogram cannot be changed, or the function or subprogram would lose the place it should return to. The SUB statement that defines an active subprogram cannot be modified or deleted until that subprogram returns to its caller. A similar restriction holds for variable allocation statements such as DIM and COMMON statements in an active subprogram. These lines that cannot be changed are called busy lines.

Even though a busy line cannot be changed, the compiled code for it may still be invalidated by an allowed change to the context containing the line. In the case of a line that invoked a multiline function, it must be recompiled when the function returns. It is clearly undesirable to have to check on every return from a function or subprogram to see if the return point is still compiled. Instead, when compiled code for a busy line is discarded, the return address in the execution stack is patched to point at an entry point

(a) Example BASIC program

```
10 PRINT "Table of Squares and Square Roots"
20 I = 1
30 IF I > 1000 THEN Done
40 PRINT I,I^2,SQR(I)
50 I = I+1
60 GOTO 30
70 Done:END
```

(b) CONTINUE command bootstraps to resume execution

```
call compiler(40)
```

(c) Line 40 compiled

```
code for line 40
call compiler(50)
```

(d) Line 40 executed and line 50 recompiled

```
code for line 40
code for line 50
call compiler(60)
```

(e) Line 50 executed and line 60 recompiled

```
code for line 40
code for line 50
code for line(30)
```

(f) Line 30 recompiled

```
code for line 40
code for line 50
code for line 30
  test for I > 1000
    true: call compiler(70)
    false: branch to code for line 40
```

(g) Line 30 executed (test was true) and line 70 compiled

```
code for line 40
code for line 50
code for line 30
  test for I > 1000
    true: branch to code for line 70
    false: branch to code for line 40
code for line 70
end program
```

Fig. 5. *Effect of interactive program modification on run-time compilation process. See text for explanation.*

Preserving Programming Investment

An important consideration throughout the design of BASIC for the HP 9000 Model 520 Computer was upward compatibility with BASIC for the HP 9845 and HP 9000 Series 200 Computers. Even though the Series 200 appeared more than a year before the Model 520, the two BASIC language systems were designed concurrently. A compatibility committee composed of members from both design teams coordinated the two efforts. As a result, Model 520 BASIC is a nearly pure superset of Series 200 BASIC. Thus, almost any Series 200 program can run without modification on the Model 520. The most significant change is usually for device select codes. The relationship between HP 9845 and Model 520 BASIC is more complex. Some features of the language were redefined to improve the consistency of the language and to pave the way for future development. The most significant changes are in the I/O and graphics sublanguages. Since not all HP 9845 programs can run on a Model 520 Computer without modification, a translator program was written to assist users in porting valuable existing software to the Model 520.

Experience to date with transporting HP 9835 and HP 9845 programs to the Model 520 has been quite good. Many programs execute successfully without modification, and most will execute correctly after manual modification of a few syntactically invalid lines. In spite of the success rate of porting programs without the translator, use of the translator program is recommended as insurance against some subtle semantic changes. There is a small set of programs that do require great effort to port. These programs contain a significant number of device-dependent portions or portions written in assembly language. Included in the device-dependent set are programs that depend heavily on directly addressing the CRT display and on certain uses of its video enhancement options.

There are three basic difference categories that the translator program handles. First is where the Model 520 supports identical semantics, but by way of a different syntax. Second is where the Model 520 supports the same syntax, but assigns different semantics to it. Third is neither of the above. Elements of this last set range from a slight change in semantics, which may affect program behavior only very infrequently, to features that have no equivalent and require user understanding of the intent of the program to make the changes. The translator recognizes almost all of these, flags them, and gives suggestions on how to translate manually.

The best example of first category is the modulo operator MOD, which has been changed to MODULO in the Model 520. Some others can result in a single line expanding to multiple lines (see MAT INPUT example below), but the semantics are still preserved.

The most pervasive example of the second category is the change from BCD to binary arithmetic. In this case the translator issues diagnostics when it sees potential problems such as noninteger numbers in FOR loop bounds and step sizes, or relational equality tests where exact equality was possible with BCD values, but will not be with binary values. A second example is the change in precedence for some operators. For example, the NOT operator has lower precedence on the Model 520 than on the HP 9835 and HP 9845 Computers.

Translation Examples

In many cases, the changed precedence does not affect the results of computations. For example, the expression −A×B means (−A)×B on the HP 9835 and HP 9845, but it means −(A×B) on the Model 520. Either interpretation of the expression produces the same answer (with the rare exception of an overflow in an intermediate result). There are cases, though, where the changed precedence does matter. The expression −3 MOD 2 yields a value of one on the HP 9835 and HP 9845 because it is (−3) MOD 2. The expression −3 MODULO 2 yields a value of −1 on the Model 520, because it is interpreted − (3 MODULO 2). After passing through the translator, the HP 9835 and HP 9845 expressions appear as (−A)× B and (−3) MODULO 2. The −A is parenthesized unnecessarily, because of simplifying assumptions in the expression parser. These simplifying assumptions are conservative—they may cause unnecessary parenthesization, but will not omit any necessary parentheses.

When the translator encounters the statement

```
20 FOR I=1 TO 2 STEP .1
```

it gives the warning

```
FOR loop with non-integer bounds or step size may behave differently due to binary arithmetic.
```

Most of the items handled by the translator could be done manually, though at the cost of considerable tedium. For example, inputting an array can be done on the HP 9845 by the statement

```
100 MAT INPUT A
```

The identical operation on a the Model 520 is accomplished by

```
100 INPUT A(*)
```

The HP 9845 also allows the redimensioning of an array by an INPUT statement, but the Model 520 does not. The statement

```
100 MAT INPUT A(3,5)
```

translates to

```
100 REDIM A(3,5)
101 INPUT A(*)
```

Finally, consider an extreme case where the HP 9845 statement

```
100 IF X>3 THEN MAT INPUT A(-N DIV M,-N MOD M)
```

is converted automatically by the translator to

```
100 IF X>3 THEN
101 REDIM A((-N) DIV M,(-N) MODULO M)
102 INPUT A(*)
103 END IF
```

If adding new lines creates duplicate line numbers in the program source, the translator issues a diagnostic, and correction of the problem will require user intervention after getting the translated source. No attempt is made to renumber existing source lines, since that would also require finding and changing any programming references to the affected line numbers.

One of the most complicated translation examples can be found in the CAT statement. The HP 9845 statement

```
100 CAT TO A$(*),Skip,N;"selector:msus",1
```

translates to

```
100 CAT ":msus" TO A$(*); SELECT "selector",SKIP Skip,COUNT N,NOHEADER
```

Note that every parameter after A$(*) in the original statement is optional. Furthermore, with the exception of the final portion (,1), each parameter is independent of all the others, and in the string selector:msus, either the selector or the :msus portion could appear without the other. In all cases the associated parameter in the translation is left out or included as necessary. The final ,1 is what causes the NOHEADER portion to appear in the translation. If this portion is ,0, the NOHEADER portion does not appear. If the

final portion is a variable, a diagnostic is given to the user to check the statement for possible manual changes.

Implementation

The translator implementation draws much from conventional compiler technology. It is driven by a recursive descent parser, which in turn relies on a scanner to build language tokens by reading the input statement one character at a time. At first glance, it appears that the translator would require complete knowledge of the HP 9835 and HP 9845 BASIC language grammar. Arithmetic expressions can occur in all sorts of strange places in BASIC statements, and every one of them must be inspected for possible changes.

The most significant simplifying assumption is that each input program is a syntactically valid HP 9845 program as SAVEd by the HP 9845's interpreter. Thus, many statements may be translated with no knowledge of their grammar. Each BASIC statement is treated as a sequence of expressions (usually delimited by a blank or a comma) which can generally be inspected and translated independently. This means that isolated keywords are processed as an expression by themselves. Complex expressions may cause recursive calls on the expression evaluator to evaluate subexpressions such as parenthesized expressions, function or procedure parameters, etc.

Of course, things are not quite that simple everywhere. Some statements must be understood in greater detail. They are handled in typical (nontable-driven) recursive descent fashion. When a keyword or expression type is detected at any level that requires more detailed analysis, a handling procedure is called, which may itself invoke the expression evaluator to handle the subexpressions. To support this detection and subsequent handling, the expression evaluator always returns the type of the expression it found and its starting and ending character positions in the source statement. This information must be kept until the statement is completely processed, since some statements require the rearrangement of many of their expressions. Which translated expression goes where depends on the type and/or existence of certain other expressions in the original statement. This kind of support is required for statements such as the CAT example earlier.

In all cases the translator tries to get by with the least understanding necessary to translate a given statement. Any primary keyword that requires no special handling is processed at the outermost level by calling the expression evaluator successively until the end of the statement is reached.

The translator itself is written in Model 520 BASIC. It contains about 4500 statements, and was designed, coded, and tested by one person in ten weeks. There were two key factors in this short development period. First, all required translator actions were well defined in advance. That is, the problem to be solved was clearly stated. Second, the Model 520 provided an excellent interactive development/debugging environment.

Acknowledgment

The help of Teresa Wall, a student summer employee, was invaluable in collecting and organizing the differences from HP 9845 BASIC.

-Gerrie L. Shults

in the static object code for the context that will set up the compiler to recompile the busy line and resume execution at the appropriate place in it.

Summary

Model 520 BASIC has the interactive friendliness of previous interpretive systems with the execution performance of a compiler. All of the interactive features of BASIC in HP's earlier desktop systems are supported.

The extra overhead introduced by run-time compiling accounts for less than 5% of the execution time of most programs and it is less than 1% for many of them. The compiling that takes place at run time is very fast since syntax is checked as lines are entered and the intermediate code produced is optimized for compiling.

For large programs, the intermediate code and object code are each about the same size as the source. (This does not include run-time support routines which are considered part of the system.) Because of the ability to throw away code when no more memory is available, a program can run (slowly) in just slightly more memory than is required for the intermediate code and variables. Furthermore, the system provides the ability to produce and execute compiled code without any associated intermediate code by using the COMPILE command.

Acknowledgments

We would like to thank other team members for their contributions to this project, which indeed was a team effort, with major contributions coming from each team member. Tom Lane contributed ideas and expertise in virtually all areas. He was personally responsible for most of the human interface design and for the internal process model. He also made major contributions to the definition and implementation of the systems programming language used to implement the system. David Wight was responsible for the early development of the intermediate code format, and later for design and implementation of the very complex executive process. Karl Freund was responsible for high-level mass storage support. He made major contributions to the design and implementation of program and data I/O to mass storage, and to the support of multiple disc formats within the single mass storage system. Gerrie Shults was responsible for alternate language support and for the MAT operations, as well as for the HP 9845 to Model 520 BASIC translator. Special thanks to the microcoders, especially Jim Fiasconaro and Bill Kwinn, who made changes and additions to the CPU instruction set to support this compiler better. Special thanks also to those who developed tools to support the development of this project. Without these tools, successful development would have been nearly impossible. The toolsmiths were Jeff Eastman, Husni AlSayed, Mike Connor, Mike McCarthy, Alan Silverstein, Dennis Georg, and Dan Osecky.

References

1. K.P. Burkhart, et al, "An 18-MHz, 32-Bit VLSI Microprocessor," *Hewlett-Packard Journal*, Vol. 34, no. 8, August 1983.
2. P.J. Brown, "Throw-away Compiling," *Software Practice and Experience*, Vol. 6, no. 6, 1976, pp. 423-434.
3. J. Hammond, "BASIC—An Evaluation of Processing Methods and a Study of Some Programs," *Software Practice and Experience*, Vol. 7, no. 6, 1977, pp. 697-712.

A Local Area Network for the HP 9000 Series 500 Computers

by John J. Balza, H. Michael Wenzel, and James L. Willits

HEWLETT-PACKARD's Manufacturer's Productivity Network (MPN) divides the computing applications for a typical manufacturing company into four areas: accounting, manufacturing, factory control, and computer-aided design. Data is collected and stored in each area and access is provided to users via combinations of computing and networking. Data access by users in the same area is required frequently and to other areas more intermittently.

In the computer-aided design area, scientific and engineering workstations are connected into clusters for resource and information sharing. LAN 9000 provides the capability to cluster HP 9000 Series 500 Computers on a local area network. In the future, additional HP-UX* workstations such as the HP 9000 Series 200 Computers will also be connected to this local area network.

Communication between the four MPN areas occurs over a backbone network. The backbone may consist of various forms of communication technology such as a local area network, packet switching, and private branch exchange. LAN 9000 can also serve as a backbone network connecting HP computers from the other three MPN areas.

Definition of LAN 9000

LAN 9000 is a product composed of both hardware and software. Its structure follows the ISO (International Standards Organization) OSI (open system interconnect) model,[1] which divides network functionality into seven layers (see Fig. 1). In the LAN 9000 implementation, the physical and link layers are accomplished in hardware, and the remaining upper layers are implemented in HP 9000 software. The physical layer provides access to the physical communications media. The link layer defines the frame format and the protocol for error detection. The internet layer provides the protocol for connecting multiple networks, multiplexing, and data segmentation and reassembly. The transport layer provides end-to-end reliability, multiplexing and flow control. The session layer provides a common interface to the transport for the applications. The presentation and application layers provide data translation and the actual network services visible to the user.

Hardware. The LAN 9000 hardware implements the physical and link layers for the Ethernet local area network specification.[2,3] The hardware consists of an HP-IB (IEEE 488) interface card connected to an Ethernet interface unit, which in turn is connected by twisted-pair branch cable to the transceiver that taps the 50-ohm Ethernet coax cable (see Fig. 2). Ethernet is a bus configuration where contention between multiple stations is resolved by a technique called carrier-sense multiple-access and collision detect (CSMA/CD). The transceiver provides the driver electronics for the cable, and the Ethernet interface unit provides address recognition, arbitration, and error detection. The Ethernet specification supports 10M-bit/s performance for up to 100 nodes on a 500-meter segment of Ethernet coax. Each branch cable can be up to 50 meters long.

Software. The LAN 9000 software consists of the upper layers of protocol and a supporting network architecture (see Fig. 1), which will be discussed later. The transport and internet levels were originally defined by the U.S. Defense Advanced Research Projects Administration (DARPA)[4,5] and are currently used in a large functional network called ARPANET.* The transport layer is called the transmission control protocol (TCP) and the internet layer is called the internet protocol (IP). The applications consist primarily of three functions: the ability to access remote files, the

*HP-UX is HP's implementation of the UNIX™ operating system.

*The LAN 9000 implementation is a subset of the DARPA protocols and has not been tested for use on ARPANET.

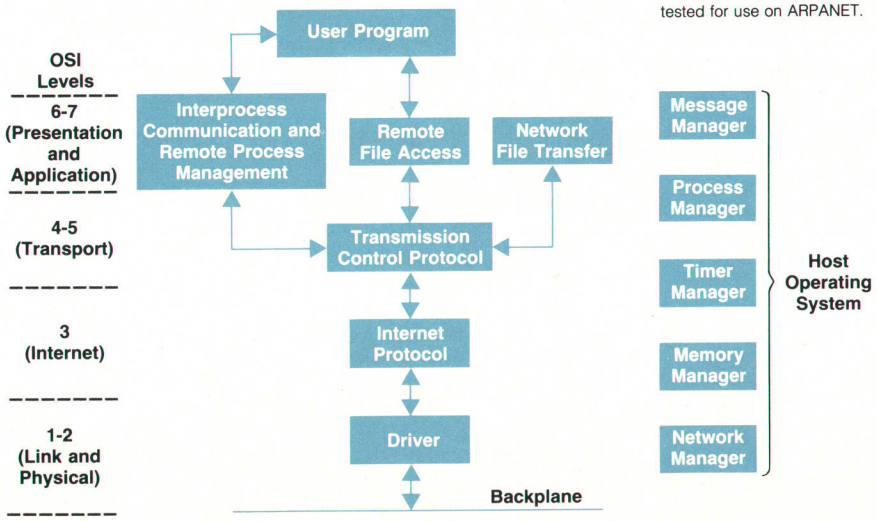


Fig. 1. *LAN 9000 software structure and its relationship to the International Standards Organization open system interconnect (OSI) model for computer network functions.*

ability to achieve high-speed transfer of files, and a lower-level tool that enables users to initiate and communicate with remote processes programmatically.

Accessing remote data is accomplished both by remote file access (RFA) and network file transfer (NFT). RFA is advantageous when accessing individual remote records and when using existing programs that access files. The method of access for RFA is a simple extension of the file path name with a remote specifier. For example, the difference in HP-UX commands between editing a local file and a remote file on node george is:

Local: vi textfile

Remote: vi /net/george/textfile

NFT is advantageous when the high-speed movement of a file from one system to another is desired. After transfer, the new file can be accessed for processing. NFT achieves about four times the throughput of RFA by using large blocks and a pipelined transfer technique. The topology for NFT is the three-node model, where the initiator, producer, and consumer can all be on different nodes. NFT is accomplished with the dscopy command, which includes the source and destination file path names as parameters. File security is invoked for both RFA and NFT by the system containing the file. Security is applied to remote access consistent with the mechanisms used for local access.

Interprocess communication (IPC) and remote process management (RPM) are lower-level tools that enable a user to write custom distributed applications. They consist of a number of procedures that can be called from the user program. RPM gives the program the ability to create and execute another program on a remote system and to terminate it. IPC consists of procedures to establish a communication path, read and write data, and terminate the path. The communication path is called a virtual circuit and enables full-duplex communication between both processes. The rendezvous between the two processes is achieved through a name assignment by one process, a name lookup by the other process, and then a handshake to establish the virtual circuit. The IPC functionality was modeled after the IPC specified in the 4.2BSD version of UNIX™ developed by the University of California at Berkeley (UCB).

Design of LAN 9000

Early in the project we knew that there would be several major problems to be solved. It was our intention to select an architecture so that as our networking needs changed, the architecture would still support them. Several key problems were recognized. First, we knew that we would be dealing with several operating systems as well as several processor families. At the time we were considering at least two different operating systems and processors, one of which was the NMOS-III VLSI 32-bit system used in the HP 9000 Series 500 Computers. We wanted to build software that could be used in any multitasking operating system with any processor family.

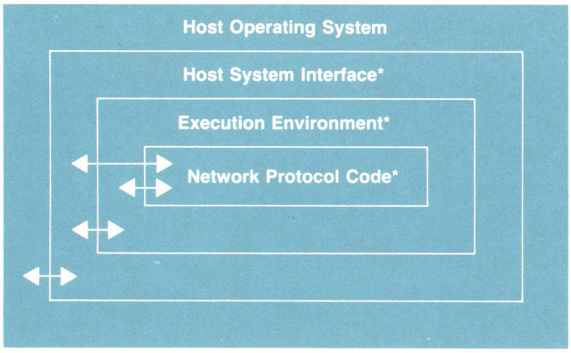


Fig. 3. *Layered isolation of portable network code.*

Second, we knew from experience that many protocols would need to be implemented within this architecture. While there are some industry standard protocols today, work in this area is just beginning. To meet HP customer needs in the future, we would have to support a variety of protocols at each of the seven levels of the OSI model. Even if we only implemented industry or international standards, there would still be a multitude of protocols, because many different physical configurations could be used to construct a network. While our initial product was only for local area networks, eventually we would need remote connections and connections over public data networks.

Third, the system had to be robust and integrated. There were several computer scientists working on the original product and over time many more would contribute to the networking functionality. We needed to define an environment where these designers could work independently and still have the result appear to be one integrated product that would be free of errors.

Because of these challenges, our first task was to define what we eventually called our data communications implementation architecture. This architecture is a comprehensive specification of module interfaces. As shown in Fig. 3, these modules are successively layered in their isolation from the host operating system. For efficiency and portability, the network protocol modules assume a very high-level execution environment that is tuned for networking code. Similarly, the execution environment

(continued on page 25)

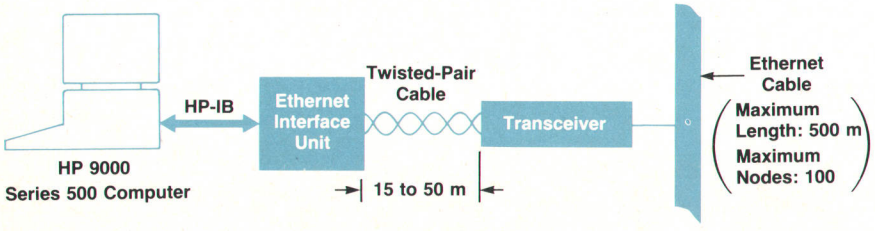


Fig. 2. *The hardware design of the LAN 9000 product implements the Ethernet local area network specification as shown.*

Data Communications for a 32-Bit Computer Workstation

by Vincent C. Jones

The HP 9000 Series 500 Computers place heavy demands on data communications. Aside from the local networking capability provided by LAN 9000, there are numerous other needs, because the real world does not consist exclusively of HP computers running HP networking software. The range of these needs is even wider than normal, because of the pivotal nature of the Series 500 itself. It needs not only the communications capability of a single-user workstation, but also those of a powerful multiuser machine.

Single-user workstations, even those as powerful as the desktop version of the Series 500, the Model 520, do not function in isolation. Effective problem solving often requires synergy between mainframe resources and the individual workstations. This requires easy communication between workstation and mainframe, particularly interactive terminal-oriented access and reliable file transfer. A typical application might require the Model 520 to offload some computation-intensive tasks from a mainframe, allowing the mainframe to provide better response to a larger number of users.

In multiuser mode, the emphasis tends to be more along the line of resource sharing among the different users. The communication link with other mainframes is a resource to be shared the same as a line printer or data base. The interactive linkup from the user's terminal to multiple mainframes is not as important a need as the ability to get required data to the user's local mainframe for processing, to communicate with users on other mainframes, or to move programs and data to larger, more specialized mainframes for processing.

A second dimension to the matrix of data communications needs is the network environment in which the mainframes operate. SNA (systems network architecture) and bisync are common with IBM host computers while DecNet™ and UNIX™ predominate on host computers made by Digital Equipment Corporation (DEC). HP's DSN services are similarly tuned to take advantage of the strengths of HP computers, while Burroughs, Univac, and just about every other computer vendor offer their own networking solutions. Unfortunately, they are all incompatible, making it necessary to implement a number of solutions while remaining hopelessly incomplete. However, IBM is such a dominant force in the mainframe market that virtually all vendors offer connection to IBM using emulation techniques. Indeed, IBM 2780/3780 RJE (remote job entry) has become so prevalent among minicomputer vendors that it is considered a *de facto* communications standard for reliable file transfer even in non-IBM environments. Similarly, almost everyone allows effective on-line access from "dumb" asynchronous ASCII terminals.

This lets us define a minimal set of communications abilities to allow efficient use of the Series 500 in most computing environments. Returning to the fundamental needs of users, we need interactive mainframe access and reliable file transfer. An asynchronous ASCII terminal emulation with programmable data rate, character size, parity, stop bits, end-of-line, start-of-line, prompt, and other parameters can be configured to access virtually any computer that can connect to ASCII terminals. By making the emulator user-modifiable (by providing source code or other techniques), access can be gained to any host that supports asynchronous terminals. Adding the capability to divert host transmission to a file and use file input in place of keystrokes provides a simple, low-cost file transfer capability. Where higher data integrity is required, IBM 2780 RJE provides a synchronous, error-controlled linking.

This leaves only interactive IBM access to provide, more commonly known as 3270 capability. Asynchronous terminal emulation can be used with black boxes known as protocol converters, but typically these are useful only under limited conditions. Most important, they are not a one-for-one replacement for an IBM 3270 display station, which requires users to memorize multistroke key sequences to access the myriad key functions available on actual 3270 systems. However, where limited or occasional access is required, especially if the user is also no longer using "the real thing," they can function quite well.

Unfortunately, IBM 3270 does not specify a unique access means. Instead, it is an entire family of products including cluster controllers, display stations, printers and integrated controller/display stations. For example, to meet varied customer needs and keep up with technology advances, there are over twenty different models of 3274 controllers (some are obsolete). There are more than ten different models of 3278 and 3279 display stations, any of which can be used with current 3274 controllers. Despite the plethora of options, however, there are really only two approaches to 3270 emulation. The first (and until late 1982, the only approach) is to emulate the entire cluster controller and attached display stations using bisync or SNA protocols to connect to the mainframe via a 370x front end. Commonly called 3274 emulation, this approach is particularly attractive for multiuser situations, where up to 32 users can simultaneously access the mainframe through the emulator while requiring only a single link from the local computer to the IBM mainframe.

The second approach, pioneered on the IBM Personal Computer by Technical Analysis Corporation (now Digital Communications Associates, Inc.), is to emulate only the display station, leaving the existing IBM cluster controller in place and hooking into the coax protocol used between controller and display stations. Commonly called 3278 emulation, this approach is most attractive when replacing individual display stations with computer workstations. Either approach, however, can typically be used in the majority of applications, albeit not always optimally. This means that the critical interconnection needs of most workstation users can be met with just three networking products: flexible asynchronous terminal emulation, simple IBM 2780/3780 remote job entry emulation, and some form of IBM 3270 capability.

In addition to these minimal requirements, other communication needs are common enough to demand specific resolution, particularly for efficient integration into HP, DEC, and UNIX environments as well as IBM.

Implementations

There are probably as many ways to develop the required capabilities as there are opinions in what makes up an adequate set of capabilities. We had choices ranging from "offer what's already available off the shelf" to "design, develop and build from scratch." As will be seen, we tried to select whatever would provide a quality product in the shortest time—typically taking an existing product and modifying it as required.

The first communications products developed for the Series 500 were two general-purpose asynchronous terminal emulators with file transfer capabilities—one for BASIC and one for HP-UX. Crucial to both was providing enough flexibility to communicate

with virtually any computer that uses ASCII characters on an asynchronous line. This means not only supporting standard options like line rates from 50 to 19,200 bits per second, 7-bit or 8-bit characters, and various parities, but also allowing options like defining what characters to use for new line and XON/XOFF host prompts before transmitting the next line, and line-oriented modes complete with start-of-line and end-of-line sequences. Also required was the ability to function with existing protocol converters for IBM 3270 and RJE.

The BASIC asynchronous terminal emulator is based on the HP 9845 Computer's high-speed terminal emulator, maintaining the same human interface so that users moving up from the HP 9845 would not have to learn a new emulator. The HP-UX asynchronous terminal emulator (**aterm**) is just the opposite, a new design from the bottom up. At the moment, the implementation is only part of the total design. Several critical features allowing modular extensions and user customization cannot be implemented until enhancements to HP-UX that will permit one process to reliably react to two concurrent asynchronous inputs are in place.

Once we were confident our minimal needs were covered, we could start looking at how to provide more specific connections. Primary criteria were timeliness of the implementation and utility to the user. This led to three main communications thrusts: HP connection via Ethernet, IBM connection via RJE, and UNIX connection via **cu** (call UNIX) and **uucp** (UNIX-to-UNIX copy).

As mentioned earlier, IBM communications consist of two major capabilities: 3270 interactive access and remote job entry. While efforts are underway to provide built-in 3270 capability, our initial effort went into file transfer via RJE. At the beginning of the project, we had to select from a number of potential options. For example, did we want to do just 2780/3780 RJE or did we want to take advantage of the multiuser capabilities of the HP 9000 Series 500 Computers and provide multileaving RJE (MRJE)? Bell Laboratories' System III UNIX, which we were building on, had an MRJE capability (the **send** command). However, that capability was built using a virtual protocol machine running on the DEC KMC-11 communications card. In addition, the System III package was based on the assumption that the only use for RJE would be to submit job streams to IBM and Univac hosts, an unacceptable restriction in view of our desire to use RJE also to exchange files with minicomputers.

Our solution was to try to take the best of both approaches; keeping the convenient job submittal facility of the System III MRJE user interface (the **send** command), but putting it atop a 2780/3780 RJE program (r2780) which could also be used directly by the user if only file transfer were required. Also required were two utility programs: a trace filter to convert card trace data from the compressed binary form generated on-line to a readable listing, and a print output filter to expand IBM carriage controls to HP-UX compatible sequences. The HP-UX standard definition for **send** is link-independent so that although the current Series 500 implementation is 2780/3780-link-based, future enhancements such as MRJE or SNA links to IBM could be added without affecting the user interface.

Third on our list of required connections, after HP and IBM, is DEC. Interactive access is fairly easy on multiuser systems—the **aterm** asychronous terminal emulator can be made totally transparent, allowing the user to take advantage of the ANSI compatibility mode offered on several HP terminals. The Model 520 workstation user is restricted, however, to "dumb terminal" only. While we consider the restriction undesirable, we do not envision many users interested in dedicating a 32-bit workstation to terminal emulation for data entry and editing. Similarly, it would be nice to hook into DecNet for file access and data transfer, but again priorities have prevented immediate satisfaction. For now, RJE suffices for reliable file transfer, even though it requires a second terminal connection to the DEC machine to control that end of the connection.

Last on our list of required connections is UNIX. Since HP-UX is based on UNIX, we felt it vital that we fit into the UNIX data communications environment. To simplify retaining compatibility with evolving releases from both Bell Laboratories and the University of California at Berkeley, we attempted to take the standard System III UNIX-to-UNIX utilities and change them as little as possible. We started out with **cu**, **uucp**, and **uux** (UNIX-to-UNIX execute). Although our goal was to leave them intact, we discovered significant design changes were required. Most critical, other than fixing numerous bugs, was removing restrictions based on the Bell assumption that all users would have source code to modify. Because HP-UX does not include an AT&T source license, features requiring modification of the source code are not acceptable unless that source code can be provided to the user by HP (i.e., was designed and written by HP, not Bell Labs). Since all these utilities are based on asynchronous dial-up links, smart modems are normally used. Unfortunately, each modem manufacturer seems to use a different protocol to tell its modem how to dial a specific phone number. Our solution was to move dialing out of the main program and put it into a separate program module, which is called from the main program and written by the user (no source license required). Sample programs showing how to dial Ven Tel and Racal Vadic modems are supplied. Similarly, in **uux** the list of programs that can be run from a remote machine was moved from a data array inside the program to an external file. Visible changes from the System III version were minimized. By retaining the original functionality and interface, standard UNIX utilities that use **uucp** still work as expected, including remote mail and the notes network.

Acknowledgments

Larry Bruns developed the BASIC asynchronous terminal emulator, Chris Fugitt wrote the HP-UX asynchronous terminal emulator, and Don Rosenbaum modified the UNIX communication utilities, including moving modem dialing out of the main program into a separate module.

Continuing our close working relationship with the I/O card developers at HP's division in Roseville, California, Rick Dow developed the r2780 program at Fort Collins, Colorado, while Brian Krelle did the I/O card firmware at Roseville. Along with them, Larry Bruns did the trace and print formatting filters, and then brought up the System III **send** command.

modules build on other environment modules and rely on the services of the host system interface, which provides a machine-independent operating system interface. The host system interface code consists of small and partially portable modules that perform whatever actions are necessary to adapt the host machine's operating system for network use. For the Series 500 operating system, called SUN (see articles on pages 28, 34, and 38), many of the host system interface functions are null, that is, straight passthroughs to system intrinsics. Ultimately, the host system interface modules could grow to constitute a small operating system in themselves when less functionality is provided by the host machine. Notice that only these modules call the host operating system directly and, therefore, they contain the

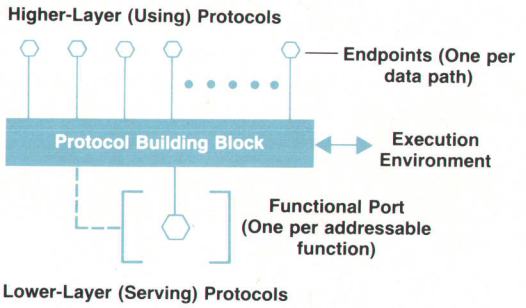


Fig. 4. *Interfaces to the protocol building block.*

only nonportable network code.

Together, the execution environment and host system interface modules provide multitasking with process synchronization, memory management and accounting, intertask message and queue management, nodal management (which controls and coordinates all the other modules in the network subsystem), utilities for manipulating the shared-memory protocol interface data structures, and a library of miscellaneous utilities (like hashing routines*) which are of general use for protocol modules.

Fig. 4 shows the logical organization of a protocol building block. This block encapsulates the code for a given protocol. The main function of the network implementation architecture is to define the interfaces between a protocol building block and the blocks above it (which use its services), the blocks below it (on which it depends), and the execution environment. The upper and lower interfaces are represented by shared-memory data structures. The actions that take place at each interface are represented by specific message types.

The lower interface for a protocol building block consists of one or more functional ports—usually one. A functional port can be visualized as a terminal strip of female electrical sockets. (The related OSI concept is called a service access point.)

The upper interface to the protocol building block consists of an endpoint for each of the protocol's instances of communication with a remote machine. An endpoint can be visualized as a male plug that attaches to a specific functional port of a higher (the using) protocol. (The OSI endpoint term refers to the following concept: Each protocol building block regards an endpoint just below it a the end of a data path "wire" that will carry its data tc peer protocol module in a specific remote machine.)

The protocol building blocks are "plugged" together by nodal management for each instance of communication with a remote machine as shown in Fig. 5. This chain of protocol building blocks is referred to as a data path and is represented as a linked list of endpoint data structures. Data paths can join or branch to represent multiplexing or alternate routing. Fig. 5 shows the data path that supports an instance of network file transfer (NFT). All data and control information related to moving a file between the local and remote machines is carried by internal messages flowing along this data path.

Note that in the current version of LAN 9000 there are several alternative building blocks at the services level. In the future, there will also be alternative modules at all the other levels as well. The protocol building block structure will allow nodal management to plug together any combination of alternative modules that is appropriate for reaching a particular remote machine. For example, the endpoint underneath the internet protocol could just as easily belong to the X.25 protocol* block, which would then be served by an endpoint belonging to the LAPB (link access protocol, balanced mode) I/O card. Also, the NFT protocol could be supported by an entirely different set of transport protocols. We have already used the nodal management capability to replace protocol building blocks by arranging data paths through alternative modules. During development, we used alternate data paths to inject special test modules at various points above or below the code being developed.

The architecture described above solved three primary problems. It isolated us from the operating system and processor set by providing a series of common function calls which we could create in any operating system. It defined a series of interfaces between protocol modules so that we could mix and match many protocols. These interfaces were based on proposals in the ANSI and ISO committees. Finally, these interfaces allowed various protocol designers to design with some degree of independence and still be sure that the system would be an integrated package.

We were concerned at first that creating all these module interfaces would cause performance problems, but that was a price we were willing to pay for the flexibility the architecture would give us. In the end, we were pleasantly surprised to find that with just a minimum of tuning, our performance was as good as or better than many other similar systems on the market. The code modularity and the architecture increased the productivity of our design group with no loss of performance.

Quality Assurance

It has long been a policy at our facility that the engineer who designed and implemented a module is responsible for the quality of that module. Following this policy, the designers wrote the test plans for their individual modules. This included both black-box and white-box testing. Here, black-box testing is based on the user manual or external specification of the module. White-box testing is based on knowing how the module was designed and stressing it at its weak points. Designers were responsible for doing their own white-box testing, and many also did their own black-box testing. The exception was when the module was designed to be used by HP customers directly. At this point, an independent tester was assigned to do the black-box testing to give us an independent opinion on the usefulness of the module.

The test plans formed the basis for determining when we were finished testing. They were also used for scheduling this phase of the project. One of the best indicators of when the quality of the product is high enough to ship to customers has been "Did we complete the test plan?" This is one reason the test plan is reviewed by the quality assurance department to ensure that it is rigorous and complete.

*Routines used to organize tables for rapid searching or look-up.

*The CCITT standard interface protocol for packet switching networks. This standard consists of three protocol layers that conform to the lower three levels of the OSI model.

The other major indicator of quality is a measure of the mean time to failure. This time is the machine time spent stressing the code in new ways, plus a derated amount of machine time spent running old test programs, divided by the number of failures detected.

Since completing the test plan usually takes some time, we used a completion estimate for scheduling this phase. Each designer estimated the hours necessary to design each of the tests in the test plan. We then calculated the amount of time necessary to find and fix code and design errors from our historical data. Finally, we allotted time for overhead and unanticipated activities. After completing these estimates for each designer, we estimated that the test phase would take about 15 weeks. Since the test plan was actually completed in 16 weeks, we felt our estimate was quite good. But at this point we still had not met our goal for mean time to failure.

In the course of doing testing, we came upon a new test method that we called triggers. Triggers is a method of triggering asynchronous events to occur at particular times. For example, if a routine asks for blocks of memory three times in its execution, we can trigger the system to reject the memory request at any of those three times. The trigger mechanism allowed us to test most of the paths in our code. It was this trigger mechanism that kept our measured failure rate so high in the beginning. Even though most of the events detected by the triggers were very improbable in real life, we continued to test for them until the triggers could not produce any more errors. Then we felt that we had a very solid system and we finally met our mean time to failure goal. This additional test time took another four weeks, but we felt the added code quality was worth the effort. Most of the problems we solved with this technique would have been very difficult to find and correct once the product was in a customer's hands.

Future Directions

The current version of LAN 9000 establishes the base to grow into additional topologies. The evolution will be in the directions of connectivity to more kinds of workstations and systems, additional links and gateways, and inclusion of more industry standard protocols. The architecture provides the flexibility to add protocols, and it facilitates the porting of the network software to other systems.

References

1. *Reference Model of Open Systems Interconnection*, International Standards Organization, ISO/TC97/SC16, Draft International Standard ISO/DIS/7498, 1982.
2. *The Ethernet: A Local Area Network, Data Link Layer and Physical Layer Specifications, Version 1.0*, Xerox Corporation, Digital Equipment Corporation, and Intel Corporation, September 1980.
3. R.M. Metcalf and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, Vol. 19, no. 7, July 1976, pp. 395-404.
4. Postel, J. (ed.), *Transmission Control Protocol—DARPA Internet Program Protocol Specification*, RFC 793, USC/Information Sciences Institute, September 1981.
5. Postel, J. (ed.), *Internet Protocol—DARPA Internet Program Protocol Specification*, RFC 791, USC/Information Sciences Institute, September 1981.

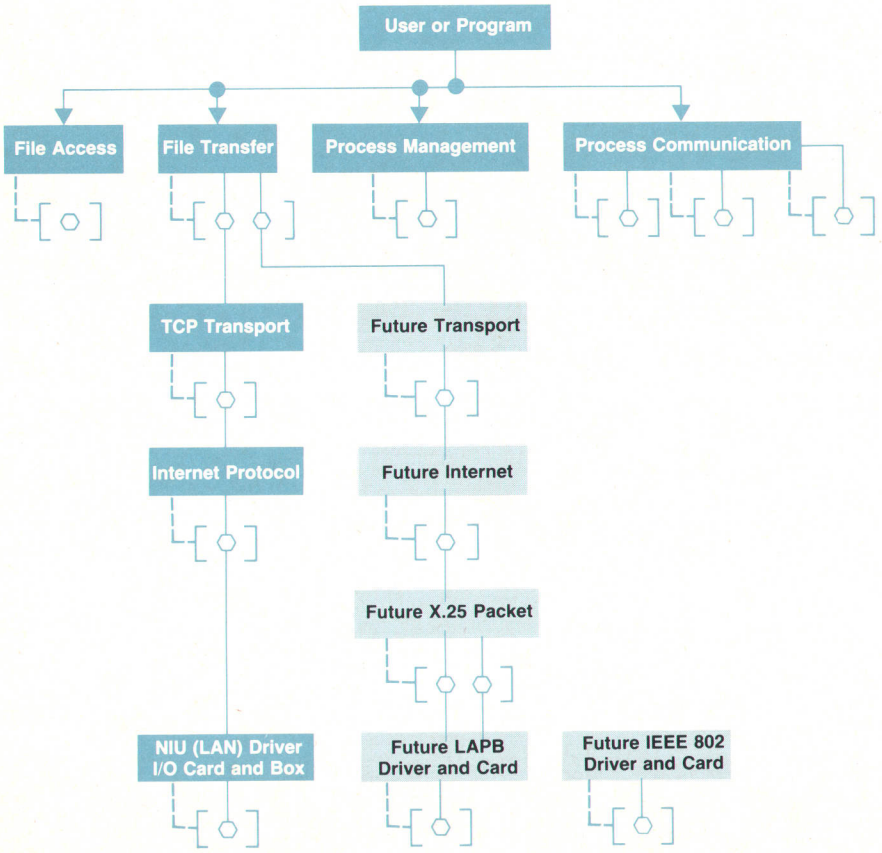


Fig. 5. *Example of a network file transfer data path formed by plugging appropriate protocol building blocks together.*

A General-Purpose Operating System Kernel for a 32-Bit Computer System

by Dennis D. Georg, Benjamin D. Osecky, and Stephan D. Scheid

THE OPERATING SYSTEM KERNEL for the HP 9000 Series 500 Computers efficiently supports the real-time requirements of the extended BASIC language environment as well as the multiuser requirements of HP-UX. The kernel provides efficient support for multiple processors, a process model that supports a large user process virtual address space, a virtual memory system that supports both paged and segmented virtual memory, memory and buffer management, and a device-independent file system which has the capability of supporting multiple directory formats. The main objective of this operating system kernel, called SUN, is to provide a clean interface between the underlying hardware and the application-level systems such as BASIC or HP-UX.

The SUN operating system can be separated into two sets of major components, as follows:

Non-I/O	I/O
Process Manager	Input/Output Switch
Memory Manager	Device Driver Modules
Buffer Manager	Input/Output Primitives
Message Manager	
Timer Manager	
Trap Manager	
Dispatcher	
Nonvolatile Memory Manager	
System Startup Manager	

The I/O components of SUN are described on page 38.

The SUN operating system manages the allocation and deallocation of hardware resources. Memory and processors are the primary system resources. Other resources include buffers, message queues, file directories, input/output channels, processors, and timers. The management of these resources supports:

- The establishment of contexts (sets of code and data addresses) for the execution of sequences of instructions
- The allocation of the processor to the execution of specific sequences of instructions
- The dynamic allocation of resources required by the algorithms being executed.

Hardware and Operating Environment

The Series 500 hardware provides a stack-oriented environment for program execution.[1] Segmentation and paging are used to facilitate memory management. A simplified diagram of the operating environment is shown in Fig. 1 on page 35.

There are two basic types of segments: code segments and data segments. Code execution on a Series 500 CPU is contained in one or more code segments and uses several data segments. One data segment is used as an execution stack segment and at least one other data segment is used as a global data segment. Each CPU contains hardware registers to define and bound the current code, stack, and global data segments. Other segments, called external data segments, can be accessed indirectly through pointers stored in the stack and global data segments. External data segments can be paged.

Information to manage the segments is kept in tables—one system segment table and many user segment tables. However, only one user segment table can be active on a CPU at a given time. The system segment table and the currently active user segment table define the address range of the program running on a CPU at any time.

A device reference table contains an entry for each I/O channel. This entry contains information to establish the code segment and global data segment for the interrupt service routine when the corresponding I/O device requests service. Each CPU has an interrupt control stack which serves as the execution stack for interrupt service routines and for the system dispatcher.

The CPU hardware defines a task control block to describe the state of a task. This block contains a pointer to the user segment table for the task and to the task's stack and global data segments. The CPU microcode uses four words of memory for each CPU in the system. These CPU-dedicated locations point to the current user segment table, the currently executing task control block, and the interrupt control stack for the CPU.

Contexts

For this discussion, a context is a set of related addresses that define a scope of addressability, that is, limit the set of code and/or data addresses that are accessible. The SUN operating system supports program, process, and partition contexts.

Program Context. The simplest context is a program, a set of one or more procedures. Each procedure is a collection of instructions, with a common entry name, which may or may not be parameterized. Instructions that make up a program are stored in code segments. A program may occupy one or more code segments or several programs may reside in one code segment. The address range (context) of a program is the set of code segments that it occupies.

During program execution, procedure parameters and local variables and an execution stack are stored in a special data segment called the stack segment. Program variables that are not local to program procedures or parameters to those procedures can be stored in either the global data segment or in arbitrary additional data segments called external data segments. External data segments are only

allocated as a result of explicit requests and can be either paged or unpaged.

The context of an executing program or process also includes the current values of the hardware registers, which define the current state of the hardware and the relative state of the process. The hardware state of a process can be established using information from the process' task control block and stack segment.

While a program has a static context, a process is an active element with a dynamic context. In SUN, a process is defined to be a unique instance of a consecutively executing program, and more than one process can share a program. The primary operational characteristic of a process is that the progress of any process in the system, as it executes its code body, is not guaranteed relative to the progress of other processes in the system.

Process Context. The minimum context for a process consists of the program context, stack and global data segments, and the current hardware state. Each process has its own stack segment. Process contexts can be expanded by the addition of an arbitrary number of external data segments. They also can be dynamically varied by allowing the executing program to switch global data segments dynamically, create and delete external data segments, or extend or contract existing segments.

Partition Context. A partition is a set of processes that share a common user segment table. This segment table has entries for the code and data segments that are local to the partition. Since the segment table entries contain the base address locations of the allocated segments as well as their current lengths, the segment table defines the segments the partition can address.

Other than the availability of memory and segment table space, there is no limit to the number of processes that can exist simultaneously in a partition. All processes within a partition can share the same global data segment. This segment represents the primary mechanism for sharing data among a set of processes within a partition.

User partition contexts are created as a result of calls to the START_PARTITION procedure. The procedure parameters specify the information required to construct a partition context as well as the context of the inital process that is to be created and executed within the created partition context. The execution of START_PARTITION allocates the initial physical memory for the partition, initializes the segment table for the partition, allocates the global data segment for the partition, and establishes the context for the initial process in the created partition. The initial process can request additional resources, or create additional process contexts. Like any other process in the system, the progress of the execution of the initial process in the created partition depends on its priority relative to other processes and the number of other processes in the system as a whole.

A partition is deleted when the last process in that partition terminates. The resources that make up the partition are then returned to the appropriate pools of available resources.

System Partition Context. The system partition is a special context defined by the system segment table. Segments described by the system segment table are addressable at all times. The union of the segments in the system segment table and the current user segment table defines the context of the machine at any time. The system segment table contains the system global data segment and other segments that can be shared by all processes in all partitions because of their global addressability.

Every process context is allocated from within a partition context. There are two classes of processes: user partition processes and system processes. The main distinction between user and system processes is the addressability of the stack for the process. The stack segments of system processes are allocated from the system segment table and are therefore always addressable. A system process can establish addressability to any partition context by changing its current user segment table, which together with the system segment table, defines the current address space. User processes cannot address any segments described in any user segment table other than their own.

Process contexts can be deleted explicitly or implicitly. A call to the SUN procedure PTERMINATE causes explicit termination of the current process. Implicit deletion occurs when the program being executed completes execution and exits its initial procedure. Regardless of whether the deletion occurs explicitly or implicitly, the effect is the same. The resources used to construct the process context are returned to the system for reallocation.

Processes in the subsystems supported by SUN always execute within the context of a partition. Process contexts established in a partition context can be used to control asynchronous events or devices, simplify the solution to an otherwise more complex problem, provide execution environments that have special characteristics such as specialized trap handling procedures, or separate the execution of subsystem-supplied code from that of code developed by a user.

An example of a process set provided by a language subsystem is the model developed by the BASIC language subsystem for the HP 9000 Model 520 Computer, an integrated desktop workstation. Each BASIC partition has access to a system human interface process and separate run and executive processes. The human interface process manages access to devices such as the Model 520's keyboard and CRT, which are controlled asynchronously by the user interacting with the machine. The executive process controls the state of the partition's run process, the parsing of language and command statements, and the communications with the human interface process. The run process performs the run-time compilation and execution of BASIC programs written by a user.

Resource Allocation and Addressability

In most cases, memory object resources are allocated from the partition containing the process making the request. For example, a request for the allocation of a data segment by a process within a user partition context results in the allocation of the segment from that partition's segment table. However, processes executing within user contexts also have an ability to allocate/deallocate memory object resources from the system context explicitly. Processes executing within the context of one user partition cannot directly allocate/deallocate resources within another user partition context.

Parallel Development of Hardware and Software

One of the earliest goals of the Model 520 Computer (the desktop version of the HP 9000 Series 500) project was to bring the completed system to the marketplace as soon as possible after completion of the hardware. The traditional project pattern in which development of the software takes place after the completion of the hardware was therefore inappropriate.

To increase both the productivity of the software development team and the resultant quality of the final system software, a high-level language was designed to be used for all systems programming. This language, called MODCAL, is based on Pascal, but includes enhancements to allow separate compilation and to provide controlled access to certain architectural features of the HP 9000 Series 500 Computers so that the temptation to code in assembly language is greatly reduced. Since the language was used to implement the most fundamental parts of the SUN operating system, it was designed in such a way as to be support-free. No supporting libraries and operating system are inherently assumed to exist by the compiler. The match between the language and the underlying architecture is further improved by the addition of a good compiler code optimizer, resulting in even less temptation to resort to assembly language.

With this strategy it was possible to develop most of the software modules which make up the system. At the end of the project more than 96% of the system software had been coded in MODCAL. This percentage included the results of extensive tuning efforts in which modules found to be critical to the performance of the machine were recoded in Series 500 assembly language. This overall stategy not only improved the productivity of the development team, but also resulted in a product with much better software reliability and maintainability.

Although it was possible to test higher-level modules by executing them on another system with simulated lower-level routines, it became apparent that the only acceptable way to check out lower-level, architecturally dependent software was to run the code in an environment that fully duplicated the characteristics of the final system. This requirement was especially critical for testing I/O driver code. Not only did it require the duplication of the CPU and I/O processor functions, but also the semantics of an I/O device.

To allow all parts of the system to complete testing and integration before the availability of functioning hardware, a detailed software emulator of the Model 520 was built. This emulator includes detailed modeling of all parts of the CPU,[1] memory controller,[2] and most significantly, the I/O processor[3] and backplane.

The HP 9845 Computer, configured with an assembly language development ROM, was selected as the engine for the emulator. The friendliness of the assembly language development environment allowed high productivity during the emulator development. The memory system of the HP 9845 was sufficiently large (500K bytes) and the overall system cost was low enough to allow several systems to be purchased for the emulation function. The last consideration was of extreme importance since the operation of the emulator is by necessity very computation-intensive and one or two copies of the emulator executing on a timesharing system would have completely consumed the system's processor. Furthermore, the HP 9845 was also used as a MODCAL development station, allowing a complete development station to exist on an engineer's desk.

The emulator was implemented in stages. The functionality of the CPU registers and a sufficient fraction of the instruction set to allow the lowest levels of the operating system to be coded and tested were provided first. This allowed most of the operating system kernel to be tested sufficiently to ready it for integration with partially tested higher-level software. Work on the emulator then continued to implement the instruction set more completely by including floating-point instructions and all instructions emitted by the MODCAL compiler. Most of the BASIC software system could be tested with the exception of the I/O drivers, file system, and human interface. A temporary I/O interface was added which allowed simple read and print I/O to take place to the keyboard and display of the HP 9845.

Next, the complete I/O processor and I/O backplane emulations were added. This consisted of software to model the state of the I/O processor connected to an external device which provided the hardware simulation of the new I/O backplane of the Model 520. Simulation of I/O device semantics could then be provided by actual I/O devices. This approach worked well for devices that were available for use, but a number of I/O devices were still under development. A capability was added that allowed software simulation of these unavailable devices.

Code was added to the emulator to capture dynamic measures of instruction and memory access mode use. The execution monitor function supported by these additions allowed the software development team to evaluate coding alternatives and to begin tuning the system before hardware was available. The functionality of the emulator was tested during its development by a battery of architectural verification tests which were developed in parallel with the emulator. These tests not only served as a cross check on the correctness of the emulator, but they were also used as verification tests for the NMOS-III chips when preliminary versions became available.

The finished emulator allowed complete integration of all components of the Model 520's BASIC system, including the human interface and I/O drivers. The execution rate of the software was 1000 times slower than real time, but was sufficient to allow BASIC statements to be stored at the rate of one every 20 seconds. At this point some of the software modules were sufficiently stable to allow the start of quality assurance testing.

Finally the hardware was ready. The 350K-byte BASIC operating system was loaded into the prototype and the system was functional. The parallel development strategy was successful.

Acknowledgments

The parallel development strategy was made possible because of the efforts of Marcel Meier and Bruce Rodean who constructed the battery of architectural verification tests and helped debug the I/O system, Eric Nelson and Craig Robinson who provided the HP-9845-bus-to-HP-9000-bus adapter, Jeff Eastman, Roger Ison, Scott Wang, Karl Jensen, Mike McCarthy, Tim Tillson, Tom Lane, Mike Connor, and others who provided the MODCAL development environment, and Bill Eads and Mike Kolesar who provided a supportive management environment.

References

1. K.P. Burkhart, et al, "An 18-MHz, 32-Bit VLSI Microprocessor," *Hewlett-Packard Journal*, Vol. 34, no. 8, August 1983.
2. C.G. Lob, et al, "High-Performance VLSI Memory System," *ibid.*
3. F.J. Gross, W.S. Jaffe, and D.R. Weiss, "VLSI I/O Processor for a 32-Bit Computer System," *ibid.*

-Benjamin D. Osecky
-Dennis D. Georg

Other types of objects can be allocated by calls to the operating system. Buffers are contiguous sections of memory that are guaranteed never to be relocated and can therefore be referenced by using absolute addresses. Buffers are mainly used by the I/O part of SUN as temporary holders of data being transferred either to or from an I/O device.

A message link is a queue receptacle to which messages can be sent and from which messages can be received. Message links are allocated by SUN from a segment that exists in the context of the system partition.

Resources allocated from the system partition have global addressability since all partitions see the system address space as part of their address space. Resources allocated in partitions other than the system partition cannot be addressed from other user partitions. The ability to address user partitions is provided to system processes via the procedure CHANGE_TO_PARTITION, which establishes user access to the specified partition.

Virtual Memory

The operating system provides support for virtual memory used in HP-UX in the form of both segmentation and paging. Virtual segments are treated as indivisible entities of variable length that can be swapped to a storage device when not in use. Virtual objects can also be allocated in paged external data segments which are divided into equal pieces called pages. Each page can reside in physical memory independent of the other pages that make up the object. Virtual segmented objects can be up to 500K bytes in size, while virtual paged objects can be up to 500M bytes.

The hardware provides indicators for each virtual object, allowing the operating system to determine if the object is currently in physical memory, and if it is, to determine whether the object has been referenced and/or modified. The operating system uses this information in its replacement algorithms to choose segments or pages to be removed from physical memory when necessary.

The operating system supports the sharing of virtual objects among several processes. It also supports the mapping of files into virtual objects, thus providing access to a mapped file at memory speeds. Virtual objects can also be locked in physical memory to prevent relocation during I/O transfers to or from the object.

Communications

SUN and the language and applications subsystems supported by it are sets of communicating processes. The initial process within the user context is free to develop an arbitrary set of processes. No structure is imposed on the process set within user partition contexts. However, all processes within a partition context share a global data segment and other segments that they can commonly address.

The simplest form of communication occurs at process creation when a single-segment relative pointer is passed as a parameter to the program to be executed in the new process context. This pointer can have no value or can point to an arbitrary parameter structure. This level of communication is similar to the parameter passing that occurs when one procedure calls another procedure within the same process context.

Processes executing within the same partition context can share data in the global data segment or other external data segments defined in their common segment table. All processes can share data in segments defined in the system segment table. Pointers to shared data or the shared data itself are stored in global data segments that are common to the communicating processes.

In addition to supporting communication via interprocess parameter passing and shared data in global data segments, SUN supports communication via message passing. This support is provided by the message manager component. The message manager supports interprocess global communication by allowing one process to construct a packet of information called a message, send that message to a mailbox, and have a different process at an later time receive the message from the same mailbox. Processes communicating via messages can exist either in the same or different partition contexts. All that is required to initiate the communication is knowledge of a common message mailbox name, which SUN refers to as a message link.

Synchronization and Scheduling

Semaphores and semaphore operations are used to synchronize and coordinate processes. A semaphore is implemented as a two-word data structure that can be allocated anywhere that can be commonly addressed by the synchronizing processes. There is no limit on the number of semaphores that can exist in the system. They are used to protect and provide exclusive access to shared data and to block a process until signaled by another process.

The semaphore operations are designed to be safe in a multiple-processor environment. By safe, it is meant that the operations on semaphore data objects are guaranteed to be indivisible and complete regardless of the number of processors in the system. A more complete description of the operation of the process synchronization primitives can be found in the article on page 34.

SUN also provides procedures for synchronizing processes with time. These procedures allow processes to wait for a specified time interval or to wait until a specified absolute time. Absolute times and intervals are specified as floating-point numbers in units of microseconds.

At any time, all processes can be divided into two groups: runable and blocked. In addition, a subset of the runable processes is actually executing on the CPUs of the system. In a system with n CPUs, up to n processes can be executing at the same time. Processes become blocked by explicitly downing a semaphore, attempting to receive a message that has not yet arrived, or waiting on a timer.

The SUN operating system supports sets of subsystems that, in general, have more processes to be run than processors. The dispatcher provided by the operating system is responsible for selecting runable processes to be executed by the CPUs based on process priority. Entry to the dispatcher occurs whenever a dispatch instruction (DISP) has been executed and the current state of the system allows the dispatcher to be entered. The DISP instruction is executed by process synchronization and manipulation primitives when the state of a process is modified. In particular, the dispatcher is entered whenever a currently executing process is blocked, when a process that is of higher priority than a currently running process is made runable, or when

A System Software Debugger

For the VLSI chip set used in the HP 9000 Series 500 Computers, two available levels of debugging evolved. The low-level capability uses a separate HP 9845 running a huge BASIC program, attached to another electronic tool, which in turn connects to the NMOS-III CPU or I/O processor debug ports. The high-level debugger is a large software package that is linked with an operating system. It uses debugging support tools built into the microcode to help programmers debug that system.

Major Features

The debugger supports stepping, breaking, and profiling at the procedure, statement, and machine instruction levels, examines and changes memory and I/O in a variety of ways, disassembles machine instructions, dumps process status, procedure chains, CPU registers, stacks, and variables, executes procedures, prints hard-copy audit trails, and passes control to user-defined debugging software.

The debugger is menu-driven. Most command prompts are less than one line long. Each command is a single letter that is accepted as soon as it is typed. Given the speed of the underlying hardware, this makes for a responsive, natural-feeling human interface which combines the best of both menu and command line styles. Novices find the debugger easy to use for quick, simple interactions. Experienced users tend to learn short command sequences that accomplish common operations.

The menus range in length from short (two options) to quite long (twelve options at the top level). Most functions are only two or three levels deep, and every menu but the top one can be exited by typing O (options).

Most menu lines are cleared after the user responds, which keeps the visual clutter to a minimum. When multiple-character input is required, the debugger requests it between menus in as compact a form as possible.

The first four options—**Pstep**, **Step**, **Focus**, and **Resume**—on the top level resume the current process, either stepping at the procedure, statement, or machine instruction levels, or resuming execution with no change of debug state. The **Break** option supports maintenance of procedure, source statement, machine instruction, memory location, and external process breakpoints. Machine instruction breakpoints can be local (one process only) or global (all processes). **Clear** sets the state of the debugger to free-run.

The **Exam** option leads to a powerful memory and I/O access capability. For examining memory, it first allows the user to specify the initial memory location in one of a number of ways, either as an absolute address, relative to various data registers and segments, or by variable name or program location. Then it supports forward and backward stepping through and jumping around memory, going indirectly through data and absolute pointers (with a return stack), modifying locations, and viewing arbitrary byte sequences. Meanwhile, the **Exam Io** option supports simple I/O requests and status displays.

The **Dump** option capabilities include task status, accumulated CPU use by processes, procedure calling chains, CPU registers, and stack and variable dumps. The **eXec** option allows users to call any procedure in memory, with a specified parameter list, under debugger control. The **Toggle** option controls debugger modes, including partial and full hard-copy audit trail printing. The **Meas** option interacts with the optional procedure, source statement, and machine instruction execution and coverage monitor (profiler).

Finally, the twelfth option, **Ud**, leads to a user-defined debugger, if one is present. Software authors can easily write their own extensions and plug them in at link time.

the priority of a currently running process is changed. This ensures that the highest-priority process in the set of runable processes is selected for execution.

The dispatcher completes the state-saving operation initiated by the DISP instruction, selects the highest-priority runable process, marks the selected process as running, restores a subset of the hardware registers based on values stored in the task control block associated with the selected process, and executes an interrupt exit (IXIT) instruction. The IXIT instruction causes the remainder of the hardware state to be restored and process execution to resume.

Interrupt Handling

The normal flow of the execution of the machine instructions can be modified by three mechanisms: external interrupts, internal interrupts, and traps. External interrupts signal requests for service by I/O devices. Internal interrupts signal abnormal conditions within the system that are not associated with the execution of a machine instruction. Traps differ from interrupts in that traps result from conditions detected by the hardware during the execution of an instruction. The detailed handling of interrupts and traps is done by the operating system.

The hardware defines 16 priority levels that can be assigned to each I/O channel. The interrupt structure is such that a higher-priority device preempts a lower-priority device. Furthermore, a special hardware register, called the mask register, can be used for the purpose of masking off specific priority levels. The initial handling of external interrupts is done by the CPU microcode interrupt handler. The interrupt handler is executed on behalf of a particular device when all of the following conditions are met: 1) the device has requested an interrupt, 2) interrupts at the device's priority level are not masked, 3) the interrupt bit in the status register is enabled, and 4) no higher-priority device is requesting service.

The interrupt handler initially saves the state of the machine by pushing a stack marker onto the stack segment for the currently executing process. The stack marker contains the information necessary to restore the status of the interrupted process, and hence allow execution to resume later. The information includes the index register value, the address of the first instruction to be executed when the machine status is restored, the machine status indication register, and a pointer to the previous stack marker.

External interrupts are handled on a special stack segment called the interrupt control stack. In this case, the CPU registers are modified to point to different stack and global data segments before the execution of the interrupt service routine. Before this register modification, the values of the registers associated with the interrupted process are saved in the process' task control block and stack segment

The debugger contains a complete user I/O facility for keyboard input and display and optional hard-copy output. Like most parts of the debugger, this code is as independent as possible of any particular operating system implementation.

Most of the debugger I/O and mode-control routines (about 34 in all) are exported to the rest of the system software. This is especially invaluable for debugging system I/O software and for supporting miscellaneous test harnesses.

The debugger includes two optional packages which can be included at link time. If the disassembler is present, all displays of code are disassembled while examining, dumping, or stepping. If the execution monitor (profiler) is present, the **Meas** option comes alive, adding a number of features.

The debugger is part symbolic. Compile-time options permit each procedure to be followed by a short symbol table, including a procedure name and possibly variable names and locations. If this information is present, the debugger uses it wherever it can. Debugging of "nondebuggable" procedures is still possible, but procedure and variable information is entered and displayed strictly by numeric address.

Implementation

This debugger is intraprocess, not interprocess. Rather than occupying one or more dedicated debugging processes that interact with others, the debugger is inactive until invoked. If the debugger is present, every process has a small amount of space (about 240 bytes) set aside at the base of its stack for permanent debugger variables. The debugger uses almost no other data storage.

When activated, the debugger runs on top of whatever process invokes it. To ensure a stable environment, it turns off interrupts, takes exclusive control of the machine (pausing all other CPUs), and is careful not to relinquish that control until exited.

The operating systems supply a limited number of special support routines to help the debugger gather information about and control other processes and operating system data structures. These routines help insulate the operating system and debugger from each other, maximizing independence.

The VLSI chip microcode provides a handy set of debugger support features. There are a number of special assembly instructions which, when enabled, cause software traps that lead to the debugger (if present). These instructions are planted in debuggable code by the compiler and enabled by the debugger on a process-local basis as needed for simple, efficient procedure and source statement stepping. Since the status register does the enabling/disabling, the debugger state becomes a part of the process state. The CPU microcode also supports machine instruction stepping and a single absolute-address break register, both of which operate through the trap mechanism.

The debugger is linked with one of a number of low-level I/O driver modules, each of which provides the same exported procedure names. These modules allow the debugger to run on an emulator, or on real hardware with the Model 520's keyboard and its various display options, or via an ASI or multiplexer card connected to a dedicated terminal.

Acknowledgments

Several people contributed to the development of the debugger. Tim Tillson developed the original concepts and first seven revisions, Dan Osecky and Bob Bury provided operating system support, Fred Richart contributed to the initial disassembler, Marcel Meier provided I/O driver assistance and contributed to the initial disassembler, and Gary Fritz developed the low-level execution monitor.

-Alan Silverstein

so that they can be restored before the process resumes.

The interrupt handler writes the device number of the device requesting service onto the interrupt control stack at a known location. The device number serves as an index into the device reference table, which contains an entry for each I/O channel. The device reference table entry for a requesting I/O device is chained by an I/O processor onto a queue corresponding to the priority of the device to await service by a CPU. Each device reference table entry contains a pointer to the interrupt service routine, and a pointer to the data relevant to that I/O channel.

The SUN operating system contains a single main interrupt service routine. Device driver procedures are executed by system and user processes as a result of input/output requests. When a device driver procedure wants to wait for an interrupt, the procedure calls a special operating system primitive which executes a DOWN operation on a semaphore associated with the device, thereby blocking the executing process. The interrupt service routine does an UP operation on a semaphore associated with the device driver procedure that handles interrupts for the interrupting device, and thus unblocks the process which had been waiting for the interrupt. The interrupt service routine executes an IXIT instruction which may force the execution of the dispatcher if the freed process is of higher priority than the currently executing process. If the unblocked process is not dispatched, then the state of the interrupted process is restored and its execution continues.

The hardware detects 45 different traps, or exception conditions. These traps are catagorized into seven classes by the operating system to make exception handling more manageable. The seven classes are system, address, program, instruction, stack overflow, trace, and debug traps. System traps include absent segment and absent page traps, and other traps that support virtual memory.

The trap manager component enables traps other than system traps to be handled by the higher-level subsystems in a hierarchical manner. Trap handling routines can be specified that apply to a specific process, to all processes in a given partition, or to all processes in the system. Trap handlers installed at the process level have the option of either handling the exception and returning to the interrupted process or referring the trap to the partition level. Similarly, partition level trap handlers can optionally refer handling to the system level.

Protection

The subsystems supported by the SUN operating system benefit from the protection of system integrity supported by the hardware as well as the protection provided by SUN itself. The protection provided by the hardware falls into three categories: segment bounds checking, mode checking, and segment attribute checking.

All segment (data or code) references are checked by the hardware to ensure that the references are within the bounds of the segment. Furthermore, any attempt to write

to a segment is checked to ensure that the segment is writable. All segments also have an attribute that indicates if the segment can be accessed by code that is designated as unprivileged. This prevents user processes from directly executing code that is strictly for internal system use. Any violations are detected by the hardware and cause traps.

The operating system provides protection beyond that provided by the hardware by providing an independent partition segment table for each user partition context. Segment accesses within a partition are limited to segments within the system segment table and to segments within the segment table local to the partition. In addition, SUN provides addressability checks at critical points in the execution of its procedures.

Acknowledgments

Several people contributed to the development of the SUN operating system. Among them, the major contributors were Marcel Meier, Bob Bury, Charlie Mear, and Bob Lenk.

Reference

1. J.G. Fiasconaro, "Instruction Set for a Single-Chip 32-Bit Processor," *Hewlett-Packard Journal*, Vol. 34, no. 8, August 1983.

The Design of a General-Purpose Multiple-Processor System

by Benjamin D. Osecky, Dennis D. Georg, and Robert J. Bury

ALTHOUGH A NUMBER of earlier Hewlett-Packard products have contained multiple-processor configurations, none has been able to bring the full power and flexibility of these processors to bear in solving user problems. For instance, the HP 9845B Computer contains an identical pair of 16-bit processors with shared memory. However, the system architecture constrains one processor to handle the computational parts of a user's program while the other processor, which accesses only the I/O bus, manages the input/output and other operating system functions. Although this partitioning of functions provides a performance advantage over a single processor for applications in which the requirements for I/O and computation are relatively balanced, the configuration does little to improve the performance of strictly computational or mostly I/O-oriented workloads.

Many other multiple-processor systems exhibit forms of asymmetry between their computation and input/output functions which are a result of either their hardware or their software architecture. On some systems a particular I/O device can be accessed by only a subset of the processors. Communication with such a device requires either complex communication protocols between the asymmetric processors or constrained execution of the user program on the processor subset. Other multiple-processor systems allow only a single processor at a time to execute operating system code.

Hardware Design

The hardware architecture of the HP 9000 Series 500 Computers has been designed to provide for a fully symmetric multiple-processor architecture. All CPUs, I/O processors, and memory controllers are interconnected by the memory processor bus. All I/O processors and memory are identically addressable by all CPUs. This implies that a program can execute on any of the system's processors without any changes to the way the system addresses either memory or I/O devices. Perhaps equally important is the fact that all I/O processors have an equally symmetric view of CPUs and memory. This makes it possible for a program to initiate an I/O operation on one processor, for the interrupt service routine to execute on the same or a different processor, and for the user program to continue on a third processor, all with complete transparency.

This symmetry is also exploited to improve system reliability. When the system is turned on, the processor components perform a self-test and report their results to one of the processors which is temporarily designated as a master. This master CPU begins the execution of operating system code that determines the number of system components that have passed their self-tests and configures the system based on these working components. Once the system has been configured, the distinction of the master CPU is canceled and the system begins normal operation, except for a possible loss of capacity caused by any failed components.

For a multiple-processor system to be able to deliver a significant performance improvement over a single processor, each processor in the system must be provided with sufficient bandwidth to the system memory. In the HP 9000 Series 500 Computers, the memory processor bus provides a bandwidth of 36M bytes per second. The memory controllers are fully pipelined and are capable of responding to arbitrary reference strings at this maximum bandwidth. Measurements of the bandwidth consumed by a single Series 500 processor indicate that the average consumption is approximately 9M bytes/s.

Another important hardware characteristic is a test-and-set operator that is atomic (indivisible) with respect to mul-

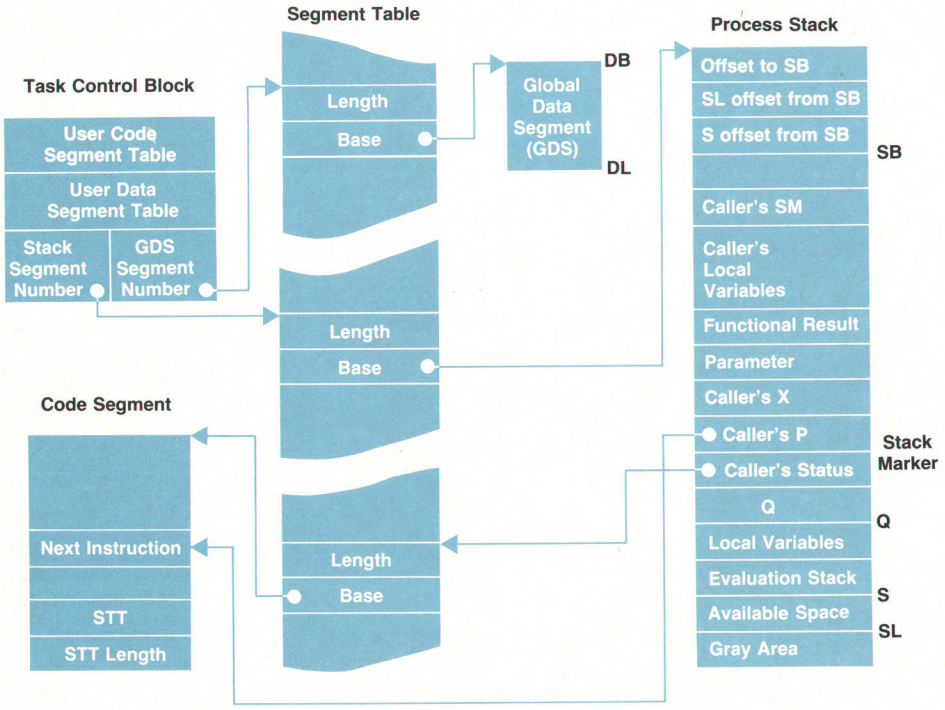


Fig. 1. *Process state as seen by the Series 500 hardware.*

tiple-processor execution. This operator is provided by the memory controller to allow the execution of a special request that indivisibly reads and sets a selected word in main memory to a predetermined value. This operator serves as a building block for constructing more complex synchronization operators in software. This hardware operator is also used by the CPUs when accessing certain table structures known to the hardware, such as page tables. This allows synchronization of access among processors, and between processors and the operating system software, called SUN, when the table entries must be modified.

Another important characteristic is the ability to share the same image of an executing program between two or more processors. This is done by providing an architecture in which all code is reentrant. Code is protected from modification by the hardware. This allows a single image of the operating system to be shared by all processors in the system and also allows several processors to be active in the operating system code simultaneously.

Finally, since the SUN operating system was designed at the same time as the hardware, it was possible to make many hardware/software tradeoffs to improve the performance of SUN. The processor instruction set provides considerable assistance by saving and restoring process states during process switching. Fig. 1 illustrates the process state known by the hardware. The task control block is selected by a processor register. Information contained in this block identifies a process' address space, stack segment and global data segment through either system or user segment table entries. The process stack contains information at its base that allows setting the stack limits and the current frame pointer. Absorbing the topmost stack frame allows information about the process' code segment and register state to be established. This entire procedure is accomplished by the IXIT instruction. This instruction and the complementary state-saving operations combine to provide fast process context switching times.

Software Design

Each process in the system has its own task control block and stack segment. The information contained in the task control block includes process state information which is known to the hardware as well as an extended process state maintained by the system software. The state information known to the hardware includes the specification of the user's address space, stack, and global data segment as described above. The software state includes the process priority, its dispatching state, fields to allow the process to be queued on a semaphore, a specification of action to be taken in the event of an exception condition, and a list of objects owned by the process.

The process priority indicates the relative priority of execution of nonblocked processes. The highest-priority process not blocked on a semaphore is always allocated a processor. If a lower-priority process is running and a higher-priority process becomes ready because of some event such as the completion of an I/O operation, the lower-priority process is suspended and the higher-priority process is given a processor. This mode of scheduling in which a higher-priority process can preempt a lower-priority process is referred to as preemptive scheduling. The system software was designed to be preemptable in all but a few small code sections.

The dispatching state indicates whether a process is waiting on a semaphore or a timer, ready for execution, or already running on a processor. The address space indicates which virtual address space contains the memory objects local to the process. System processes often coexist within the same address space and communicate directly using shared-memory techniques. User processes for both

HP-UX and BASIC systems each exist within their own address space, which can be up to 512 megabytes. Every process also has access to the system address space, which can also be as large as 512 megabytes.

Three different mechanisms are provided to allow the synchronization of process activities: locks, semaphores and message passing. A lock provides the short-term exclusion mechanism normally provided by disabling interrupts while in a critical section on a single-processor system. This same effect is accomplished on a multiple-processor system by performing a special code sequence on a memory word, referred to as a lock word, that is associated with the critical section in question. The exclusion operation is performed by executing the instruction read and set to −1 and testing the result. If the result is not equal to minus one, the lock has been obtained and the critical section can be executed. If the lock value is already minus one, the processor must retry the indivisible read and set to −1 instruction until a nonnegative value is read. It is then assured that no other processor is active in the critical section. When the processor has reached the end of the critical section, it stores a zero back into the lock word to indicate that the critical section can now be entered by another processor. This locking mechanism is used many places in the system where exclusion is required for a code sequence that is short and does not execute any operations that cause the process to be blocked.

A more general process synchronization tool is provided by semaphore operations. The implementation of semaphores is similar to that proposed by Dijkstra.[1] A semaphore is a two-word area in memory that contains a value and a pointer to a linked list of processes. Semaphores can be allocated in memory anywhere that other data structures can be allocated, and there is no limit on the number of semaphores. Two operators are provided for semaphores: DOWN and UP. A DOWN operator applied to a semaphore with a value greater than zero merely decrements the value associated with the semaphore. If the value is less than or equal to zero, the value is still decremented, but the process' state is marked blocked and the process' task control block is added to the queue of waiting processes associated with the semaphore in order of priority. The UP operator increments the value of the semaphore. If the initial value of the semaphore is less than zero and an UP operation occurs, the first process waiting on the queue is marked as ready. If the process marked ready is of higher priority than the process executing the UP operator, the processor is given to the higher-priority process.

Serialization can be provided for a critical section by associating with it a semaphore initialized to a value of one and providing a DOWN operator at the beginning of the critical section followed by an UP operator at the end of the critical section. Synchronization with a server process is usually accomplished with a semaphore initialized to a value of zero.

The word used as the head of the queue of tasks blocked on a semaphore also serves as a lock word to guarantee proper operation of the semaphore in a multiple-processor system. Since there is a separate lock word for every semaphore in the system, the probability of contention for the lock is very low.

During development of the software, it was found that considerable simplification would result by including additional semaphore operators. The first consisted of a conditional UP operator which would free all processes waiting on a given semaphore and allow the passing of an error escape code. This operator is especially useful in recovering from an error condition detected by another process. The second consisted of an indivisible DOWN and UP operator which would allow a process to block itself on a semaphore while releasing another semaphore in one indivisible operation.

Message passing supports interprocess global communications by allowing a process to construct a packet of information called a message, send that message to a mailbox, and have a different process at a later time receive the message from the same mailbox. Message passing and mailboxes are used by both BASIC and HP-UX system processes to coordinate user processes. Sending and receiving messages provides process synchronization similar to the semaphore UP and DOWN operations, coupled with an address-space-independent data transferral. The message passing operations, in fact, are implemented using the semaphore operators, which in turn use the locking concept at the lowest level.

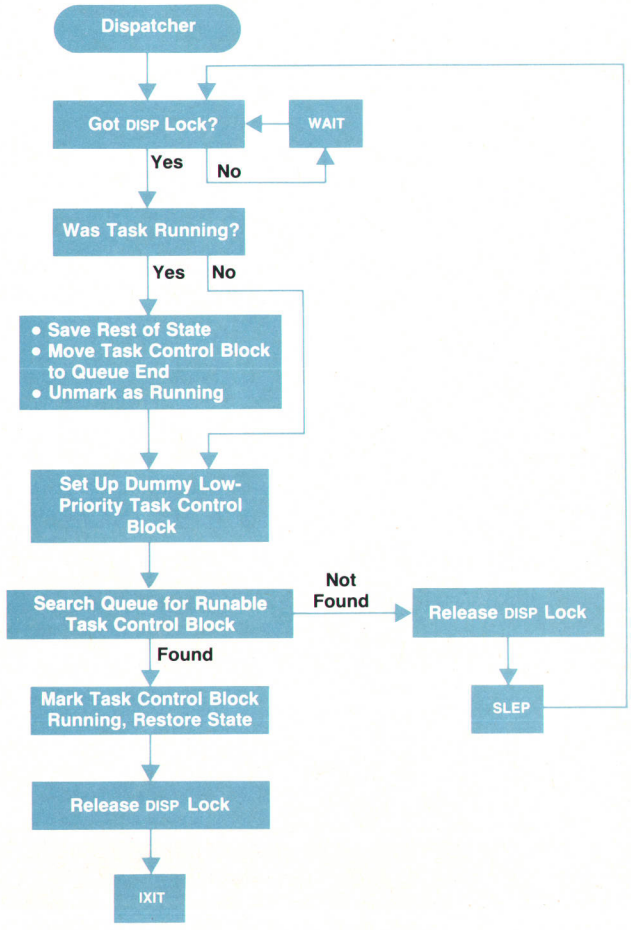


Fig. 2. *Dispatcher flow chart.*

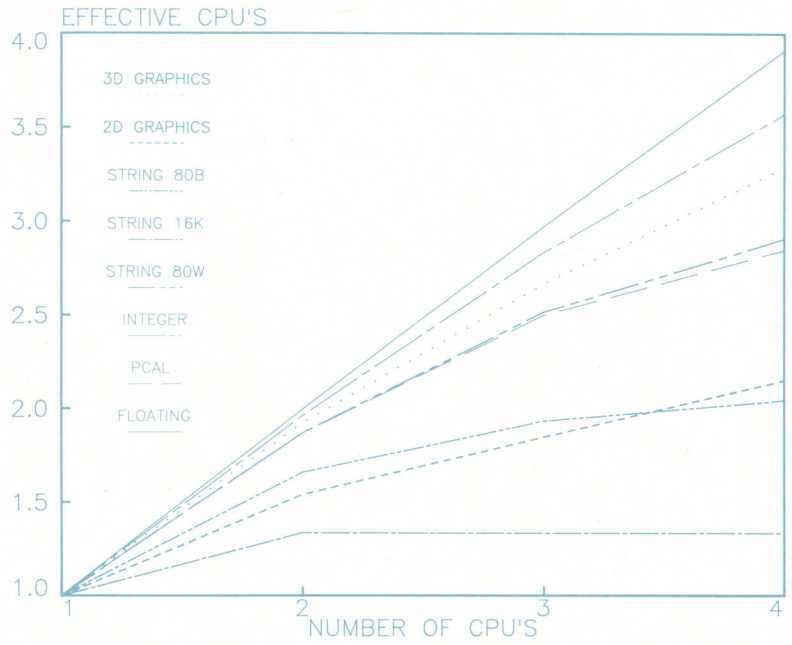


Fig. 3. *Multiple-processor performance with a homogeneous load. Each benchmark run consists of multiple identical copies of the same program.*

Granularity*

The duration of a typical process lifetime in HP-UX, which can last from a few tens of milliseconds to forever, is well matched to the granularity of the underlying process model. The BASIC system's process model similarly is of reasonably large granularity. The times for representative operations for the underlying process model are:

Operation	Time (μs)
Process Creation	1000
Lock Type Synchronization	5
Semaphore Operation	35
Process Switch	150
Assign Free Processor	100

To illustrate how the overall process model is implemented, consider the flowchart of the dispatcher shown in Fig. 2. When a DOWN synchronization operation dictates a process block, the process is marked blocked and the special instruction DISP is executed. This causes the process state to be saved and control to be transferred to the beginning of the short-term scheduler routine at the dispatcher entry point. Upon entry, the dispatcher ensures that just one processor at a time is active within the critical section of the dispatcher by attempting to lock a word

*In this article, granularity is a measure of the size of independent operations that a process or a program can be broken up into for parallel processing. In each case, the performance benefits gained by parallel processing must be weighed against the system overhead required to break up the process or the program. That is, finer granularity requires more overhead.

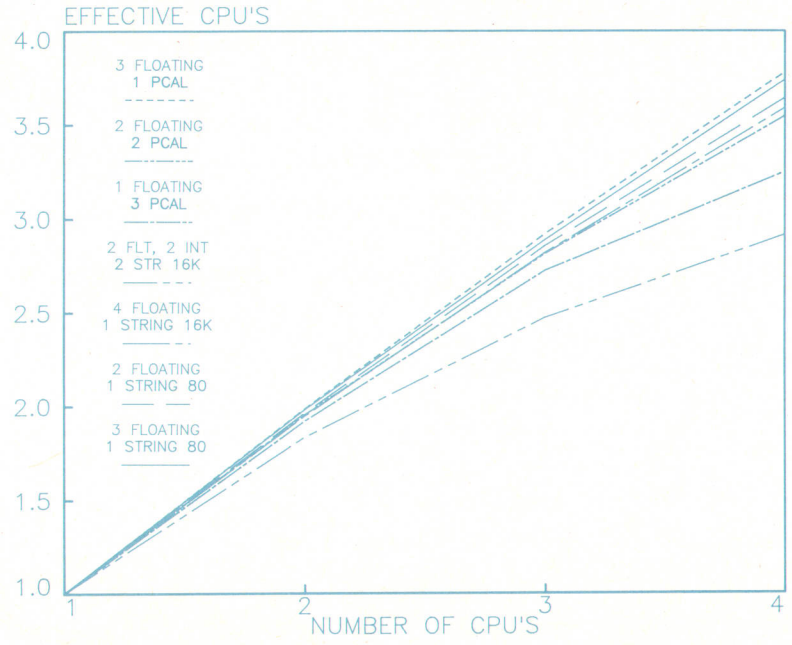


Fig. 4. *Performance of benchmark runs containing nonidentical process loads.*

associated with that critical section. Once inside the critical section, the dispatcher scans the linked list of available processes using a special hardware linked-list search instruction. The highest-priority process that is not blocked and is not currently running on another processor is selected for execution. The dispatcher's lock word is released and the process is launched by setting the the CPU's current task control block pointer to point at the selected process. The context switch is completed by executing an IXIT instruction as described earlier. This results in the restoration of the state of this task and its continued execution until it is either blocked or is preempted.

If no tasks are found that are available for execution, the dispatcher's lock is released and the special instruction SLEP is executed. SLEP places the processor in a state where it is paused until the next interrupt or interprocessor message. In this way processors that have no work to do consume no bus bandwidth, yet are prepared to respond quickly when an event indicating the possible presence of a new runable task occurs.

Performance

The final test of the design of any multiple-processor system is in how much improvement in performance is provided by each additional processor. Fig. 3 shows the results of a number of benchmark runs which were made by running four copies of an identical benchmark on each of four processor configurations. The programs were selected to show the performance extremes that can be encountered in a multiple-processor system load. The programs showing the least improvement were STRING 16K and STRING 80B. These programs make use of the processor's block-move instructions, which are capable of moving data from one place in memory to another at the rate of 9M bytes/s. Large and repeated block moves place a heavy load on the memory processor bus.

The two-dimensional graphics benchmark is mostly I/O-bound on a single processor and adding additional processors does not produce a very dramatic result. The other benchmarks, 3D GRAPHICS, INTEGER, PCAL, and FLOATING, are respectively a three-dimensional graphics program with I/O, an integer matrix multiply program, a very heavily recursive program, and a floating-point matrix multiply program. All of these loads are significantly improved by having additional processors in the system.

Fig. 4 shows the results of tests in which groups of nonidentical processes were run on systems containing a varying number of processors. The results from these runs are more uniformly clustered near the theoretical n-improvement-for-n-processors asymptote. This indicates that a significant improvement is possible even for loads including bandwidth-intensive programs such as STRING 16K as long as they are included with programs containing more typical instruction mixes.

Acknowledgments

We would like to acknowledge the efforts of Mike Kolesar and Jim Fiasconaro whose careful attention to the design of the multiple-processor aspects of the hardware made life much easier for the software team. We would also like to thank Bill Eads for letting us work on this project.

Reference

1. E.W. Dijkstra, "The Structure of the THE-Multiprogramming System," *Communications of the ACM*, Vol. 11, no. 5, May 1968, pp. 341-346.

An I/O Subsystem for a 32-Bit Computer Operating System

by Robert M. Lenk, Charles E. Mear, Jr., and Marcel E. Meier

MEETING THE DIVERSE NEEDS for input/output processing for both the BASIC and HP-UX subsystems in the HP 9000 Series 500 Computers posed a significant challenge during the design of the operating system called SUN. The BASIC language system for the Model 520 Computer provides a rich I/O language with support for real-time device and instrument control. HP-UX is a multiuser system which relies very heavily on rapid access to disc storage for loading user programs, storing user data, and managing virtual memory. In addition, both systems provide users with a unified, device-independent I/O interface to all peripheral devices and mass storage files. SUN's I/O subsystem provides common code that fully supports the needs of both.

The SUN I/O system consists of two primary software components—the file system and the device drivers. The device drivers provide a uniform high-performance interface for managing peripherals while the file system provides file management, disc memory management, and device management services to other components of the operating system as well as to user environments such as HP-UX and BASIC. In general, the structure of the I/O sub-

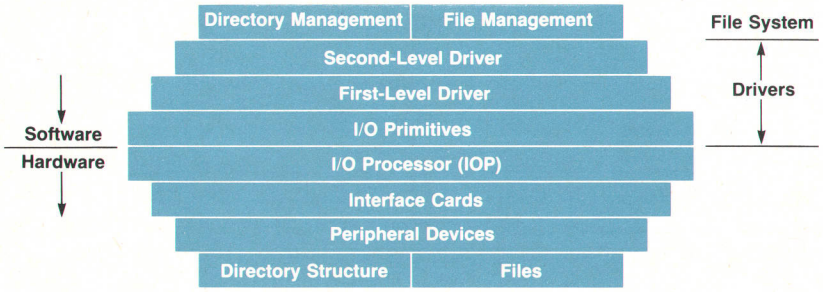


Fig. 1. *The SUN operating system's I/O subsystem.*

system (Fig. 1) mirrors the functionality of the hardware.

File System

The basic unit of disc storage managed by the file system is called a volume. Each volume has a slave process that performs all the I/O to the volume. The creation of a separate, transparent process allows physical I/O to be performed concurrently with other tasks and provides a mechanism whereby I/O requests can be scheduled in the most efficient manner possible. Volumes have self-contained data structures upon which files and directories are implemented. Directories, which map filenames into files, are managed solely by the file system while the interpretation of the contents of files is left to higher-level software.

Multiple Disc Formats. The file system supports file management of three different disc formats: HP's Logical Interchange Format (LIF), the disc format used by the HP 9825, HP 9835, and HP 9845 family of computers, and the Structured Directory Format (SDF). The support for multiple disc formats was motivated by the desire to provide file interchange capability with other HP systems, to access discs initialized on earlier HP computers (backward compatibility), and to provide additional capability not supported by either the LIF or HP 9845 formats. The support for multiple disc formats does not introduce additional overhead to the file system.

A distinct software module manages each of the three formats. A common interface hides the disc format differences from other file system software and the BASIC subsystem. When a disc is first accessed, the file system identifies the disc by the contents of block 0 on the disc and installs the correct software module to manage its structure. The file system returns an error if the caller attempts to use a feature not supported by the particular disc format, but otherwise no distinction between formats can be made by the application. Because of its extended capability, HP-UX supports only SDF as its root file system. However, HP-UX applications can access LIF discs through standard utility programs.

SDF supports capabilities beyond those of the other two disc formats. Among them are extensible files, hierarchical file naming, file links, extensible directories, mounted volumes, device files, remote file access support, and HP-UX file protection mechanisms.

Each file on an SDF volume is described by a 128-byte file control block (FCB), similar to the UNIX™ inode. The FCB contains information about where the file resides on the disc, when it was last created, accessed, and modified, and how its use should be restricted. Disc space is allocated to the file in contiguous areas called extents (identified by the address of the first block of the disc area and its size). This form of representation enables large, high-speed transfers between the disc and the memory, supports large files efficiently, and allows the amount of disc space allocated to the file to change dynamically.

To support the needs of both the BASIC and the HP-UX systems, a portion of the FCB is reserved for private use by the subsystem that created the file. HP-UX, for example, uses this private data area to implement device files and HP-UX-style file protection semantics.

Caching for Improved Performance. The file system uses a pool of equal-size buffers (buffer cache) to improve disc access performance. When data is read from the disc, it is placed and kept in the buffer cache. When a subsequent request is made for the same data, it can be retrieved from the cache without requiring any physical I/O operation. Accessing data in the cache is more than an order of magnitude faster than obtaining the same data from the disc.

The number of buffers in the cache is determined when the system is initialized. Eventually the contents of one or more buffers has to be discarded to read new data from the disc (data not found in the cache). In this event, the cache buffer that has been least recently accessed is chosen.

The cache is also used to improve performance in other ways. When sequential access to a file is detected, the file system prereads data from the file in anticipation of the next read request. The data is then kept in the cache until it is needed. The I/O required for the read is performed concurrently with the running of the application program. By prereading data, the file system overlaps CPU processing with I/O device time, thereby reducing the total time it takes to run an application.

A large portion of the time needed to move data to and from the disc is spent waiting for the disc to prepare for the transfer. The actual transfer of data takes much less time. The file system transfers as much data as possible on each disc read/write. When prereading data, several buffers of data are read from the disc with one transfer. The cache also allows a file's dirty buffers (buffers that must be written back to the disc because their contents have been modified) to be gathered together and written to the disc in a single transfer.

UNIX is a U.S. trademark of Bell Laboratories

Virtual Memory Support. Since many HP-UX systems are configured with only one disc, the file system must handle file management and virtual memory support together on a single volume. Each disc format module has entry points that the virtual memory system uses to allocate and deallocate disc blocks. These same high-speed disc space management routines are used by the file system to allocate disc space for directories and files. Disc storage is fully shared between the file system and the virtual memory system. The physical I/O generated by the virtual memory system as a result of paging and segment swapping does not move through the cache, because the virtual memory system does its own caching through sophisticated page and segment replacement algorithms. This I/O is sent directly to a volume's I/O process. The file system and virtual memory system together support the concept of a memory-mapped file. Files can be mapped onto a virtual segment and accessed through a pointer. Once mapped, the virtual segment contains an image of the file. Changes to the address space represent changes to the file and vice versa. This form of access is sometimes more convenient than standard file access routines and can, in certain applications, result in improved file access performance.

Transparent Device I/O. Transparent device I/O for HP-UX is supported through device files, which contain information about the location of a device (possibly logical) in the system, and the manner in which the device and system must communicate. These special files are opened and accessed in the same manner as other files. Their I/O, however, is directed to the appropriate device.

Drivers

The underlying philosophy behind the driver architecture is that each piece of hardware is encapsulated by a separate module. A typical I/O operation involves three separate pieces of hardware: an I/O processor, an interface card, and a peripheral device.[1] Thus, the driver implementation includes modules in three layers—I/O primitives, first-level drivers, and second-level drivers. This allows drivers to be mixed and matched for appropriate tasks without duplicating functions. Thus, all peripherals, whether discs, tape drives, line printers, or voltmeters, can share the same HP-IB (IEEE 488) interface card first-level driver, while a single CS-80 protocol second-level driver can suffice both for HP-IB-based discs and the internal discs on the Series 500's integrated workstation, the Model 520 (see Fig. 2). Because of this modularity, the potential also exists to move any first-level driver to other machines where the same interface cards are present on different I/O channels, or to move any second-level driver to other machines where the same peripherals use different interface cards.

Each of the three layers invokes the one below it through a procedure call. This keeps the overhead of the modularization to a minimum. However, in special cases where even this overhead is considered excessive, an individual driver module crosses these conceptual layers to optimize performance. A primary example of such an optimization is a special driver to do fast reads and writes of CS-80 discs on the HP-IB interface.

The modular driver organization allows software to be configured to match precisely the hardware on which it runs. A minimal software system contains only the I/O primitives and the drivers necessary for a minimal hardware configuration without wasting kernel code space on unnecessary modules. As more hardware is added, the corresponding drivers are added to the software. All global system tables of drivers and devices are built dynamically by the modules that are present at system boot, rather than being compiled into a central part of the code or requiring a complicated system generation activity. Hence, the addition of drivers does not require any recompilation or relinking of the system; it is accomplished by simply merging the driver code into the system boot area in HP-UX and rebooting, or by executing the LOAD BIN command in BASIC. The same ability to configure modules into the system is used by the file system for the modules that manage different disc formats, and by other subsystems outside of I/O.

I/O Primitives. The primary purpose of the I/O primitives module is to encapsulate the interface to the I/O processor and the services it provides. These services include direct memory access (DMA) transfers across the backplane, passing interrupts to the CPU, and running channel programs (lists of I/O operations which are run by the I/O processor without CPU intervention). These services are presented to the drivers as routines that are independent of the nature of the I/O processor, such as setting up a DMA transfer or waiting for the interrupt at its completion. A few of these routines that are simple but frequently executed are implemented with special compiler support by in-line expansion in the calling driver's code.

There are other tasks which, though not directly related to the I/O processor, are common to several drivers, and thus are also included at the primitives layer. For example, the primitives module examines all I/O slots at system initialization to determine which interfaces are present. This is an essential part of the self-configuration process, because it allows each first-level driver to select which interface cards are appropriate for it to address without any knowledge of the behavior of other cards that may be present.

The primitives also provide for resource allocation among drivers to prevent multiple requests from interfering with one another. This is generally handled by providing mutual exclusion to each I/O slot, but it also involves the length of request. For shorter requests, interfaces such as terminal multiplexers allow multiple outstanding requests with certain restrictions, which are enforced by the primitives. For longer requests, each I/O processor has

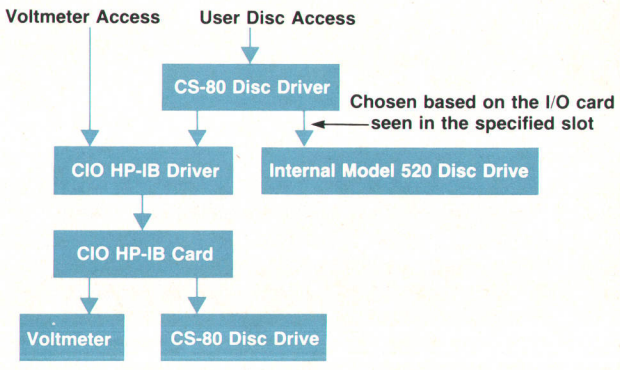


Fig. 2. *Driver structure for HP 9000 Model 520 Computer.*

bandwidth limitations, and in rare instances it is possible for a high-speed device to lock out a synchronous device on a separate slot. The primitives provide mutual exclusion between such incompatible devices on the same I/O processor.

HP-CIO. The HP 9000 Series 500 Computers are the first HP products to support HP's new family of interface cards, known as HP Channel I/O (HP-CIO). Each card in this family shares several levels of protocol, some of which communicate fairly complex tasks between the host computer and a microprocessor on the card. The card's microprocessor can perform such tasks as searching input streams for a termination character, or editing lines of text input from a terminal. Much of this protocol is encapsulated at the primitives level, allowing not only a sharing of code among drivers, but also efficient implementation of the protocol by matching it carefully to the I/O processor's functionality.

The I/O primitives level provides a useful layer to insulate the drivers from the I/O processor. Hence, all the interface card and peripheral drivers can be written in MODCAL (HP's internal Pascal-like systems programming language) rather than being forced to use assembly language. This encapsulation of the assembly language in the I/O primitives reduces the time required to design, code, test, and maintain the driver compared to programming in assembly language. The reliablity of the drivers is greatly improved, because it is easier to understand the code and its function when the code is in a high-level language.

First-Level and Second-Level Drivers. The first-level and second-level drivers are designed to hide the anomalies of the peripherals and interface cards while providing all the functions that each device provides (e.g., full access to the instrumentation features of the HP-IB interface). Nonspecific device features are relegated to higher levels of the I/O hierarchy to prevent duplication of functions that would increase the overall size of the I/O subsystem, reduce performance, or possibly present inconsistent behavior for different drivers. In addition to encapsulating the specific peripheral or interface card characteristics to provide access to a generic device, the driver design provides access to rather dissimilar devices (e.g., discs and HP-IB interface cards) with the same parameters for either the first- or second-level driver procedures. This uniformity provides the first step in supplying the user with a totally device-independent I/O interface.

The SUN operating system drivers also provide extremely resilient recovery from error conditions. They have been through a thorough set of tests to ensure that the drivers never leave the device they control or leave the system in a bad state (requiring a power cycling of the computer or device) as a result of any possible error condition. Extra effort was taken to provide a broad resolution of error conditions reported to the system rather than combining many different errors into generic error values. The drivers also provide numerous different soft-error reports to inform the user of nonerror related data such as the occurrence of an automatic data record sparing (replacement of a bad record with a good record) on a mass storage medium or a case where the system had to retry an access to a data record to obtain it without an error. This additional resolution and the soft-error concept provide the user with a more informed view of the operation of the system instead of requiring a guess as to what is going wrong.

HP's earlier desktop computers have always provided access to the hardware registers on the I/O cards to provide customers with access to features not provided by the I/O language of their system. However, the new-generation HP-CIO cards are far more complex to program and in addition, do not have conventional registers. Thus, the SUN drivers provide a synthesized set of registers, called pseudoregisters. This provides the user with the same model as on the HP 9845 Computer for accessing features of an I/O card not normally provided by the system, but done in cooperation with the driver. Thus, the driver has the opportunity to provide additional functions in an isolated manner. This allows the same pseudoregisters to be used for different implementations of the same type of interface card, which increases the portability of user applications.

The SUN drivers support the broad set of peripherals produced by Hewlett-Packard. Instead of initially supplying a core set and then adding less-essential drivers at a later time, SUN provides support for all peripherals that make reasonable sense for the HP 9000 marketplace. One additional driver that is somewhat unusual is the memory driver. This driver uses the main memory of the running system to simulate a disc drive. The code required to provide this driver is a great deal simpler than if this function were provided at a higher level. The result is that a user can develop an application that accesses a disc file and then decide to store the data in main memory to gain the performance advantage of not having to spend the time to access a disc drive. The change is trivial using the memory driver; simply revise the reference to the specific mass storage device to be the memory driver rather than the original disc drive.

Acknowledgments

Gary Fritz did the initial implementation of the I/O primitives, David Frydendall implemented the serial and 16-bit parallel card drivers, Mark Hodapp tested the file system, Dan Osecky assisted in the implementation of the I/O primitives, and David Pinedo tested the HP-IB and nine-track tape drivers and implemented the CIPER protocol printer driver.

Reference

1. F.J. Gross, W.S. Jaffe, and D.R. Weiss, "VLSI I/O Processor for a 32-Bit Computer System," *Hewlett-Packard Journal*, Vol. 34, no. 8, August 1983.

Authors

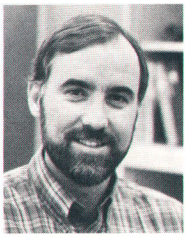
March 1984

3

Michael L. Kolesar

Mike Kolesar studied physics at Villanova University (BS 1968) and nuclear physics at Cornell University (MS 1971). He came to HP in 1974 with three years of experience in designing high-speed data acquisition systems for a synchrotron laboratory. Now a section manager responsible for graphics software and HP-UX commands and languages, he contributed to the architecture and microcode for the Series 500 CPU chip and managed some of the Series 500 software groups. He is coauthor of an award-winning paper on the Series 500 CPU architecture. Born in Chicago, Illinois, he is married, has three daughters, and now lives in Fort Collins, Colorado. He enjoys downhill and cross-country skiing, stereo music systems, photography, electronics, and hiking in the Rocky Mountains.

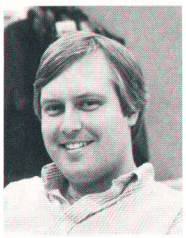
Michael V. Hetrick



Mike Hetrick began work at HP in 1973 and contributed to the production of the HP 9830 Computer and development of the HP 9815 and HP 250 Computers. He was project manager for the data base management and operating system portions of the HP 250. An R&D section manager since 1979, he has been responsible for the development of the high-speed HP 9845 Computer, the Model 530 and Model 540 Computers of the Series 500, and most recently, the first HP-UX products for the HP 9000 computer family. Born in Milwaukee, Wisconsin, Mike studied electrical engineering at General Motors Institute (BSEE 1969) and the University of Colorado (MSEE 1970). He is married, has two sons and two daughters, and lives in Loveland, Colorado. He enjoys racquetball, softball, and camping, and plays clarinet in the Loveland Municipal Band.

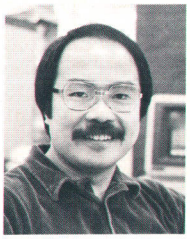
7

Jeff B. Lindberg



Joining HP in 1976 with a BSEE degree awarded by the University of Nebraska, Jeff Lindberg worked on operating system design for the HP 250 Computer and the HP 9000 Series 500 HP-UX product, for which he is now project manager. He received an MS degree in computer science from Colorado State University in 1982. Jeff is married, has a son, and lives in Fort Collins, Colorado. Outside of work, he is a vocal soloist and enjoys playing golf and basketball and spending time with his family.

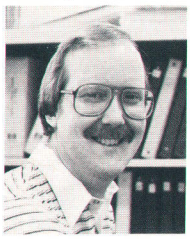
Scott W.Y. Wang



Scott Wang studied electrical engineering at the Massachusetts Institute of Technology (SBEE 1971) and the University of Michigan (MSEE 1972). With HP since 1972, he has contributed to a number of HP products, including the HP 9805 Calculator, the HP-81 Computer, and the 16K NMOS ROM. He managed the development of the HP 250 Computer's BASIC operating system and OM250 Applications Pack. He managed the HP-UX project and the software tools for the Series 500 and now is responsible for developing HP-UX for the Series 200 Computers. Scott is a member of the Computer Society of the IEEE and lives in Fort Collins, Colorado. He is married, has a daughter, and is interested in audio/video, home computers, and photography.

15

Timothy W. Tillson



A software development engineer at HP's Fort Collins Division, Tim Tillson worked on the BASIC compiler and human interface for the Series 500 Computers. Tim is a member of the ACM and holds an AB degree in mathematics from Brown University (1974) and MS degrees in mathematics and computer science from Ohio State University (1976 and 1978, respectively). His studies resulted in three research papers related to combinatorial mathematics. Born in Houston, Texas, he now lives in Fort Collins, Colorado. He is married (his wife is an English professor) and actively interested in running, swimming, reading, software hacking, and the stock market.

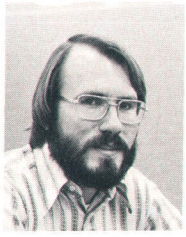
Richard R. Rupp

Dick Rupp joined HP in 1979 after receiving a BS degree in computer science from Michigan State University. He was a software engineer working on the BASIC front end for the Series 500 Computers before leaving the company recently. A native of Detroit, Michigan, he is single and lives in Denver, Colorado. He likes playing volleyball, water skiing, and working with stained glass.

Jack D. Cooley

A native of Mattoon, Illinois, Jack Cooley attended the nearby University of Illinois, earning a BS degree in physics in 1966.He served four years in the U.S. Air Force, attaining the rank of captain, before continuing his studies at the University of Colorado. He received an MS degree in computer science in 1972 and then joined HP. Jack has worked on a number of BASIC language projects for various HP products, including the HP 9835, HP 9845, and Series 500 Computers, the last as a project manager. He is now a software engineering manager for product assurance at HP's Fort Collins Division. He is married, has two children, and lives in Fort Collins, Colorado. Outside of work, he plays folk guitar and enjoys running, hiking, camping, bicycling, cross-country skiing, and photography.

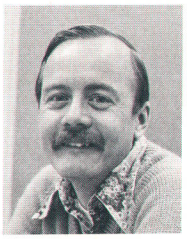
David M. Landers



Dave Landers grew up in Indianapolis, Indiana, and attended Purdue University, receiving a BSEE degree in 1973 and an MSEE degree in 1974. He then joined HP and has worked on BASIC software for a number of HP computers—the HP 9835, HP 9845A, and HP 9000 Series 500. He also contributed to the I/O ROM for the HP 9835 and HP 9845 and to the LAN 9000 software. Dave is single, lives in Fort Collins, Colorado, and enjoys chess, hiking, cross-country skiing, and playing softball and basketball.

22

John J. Balza

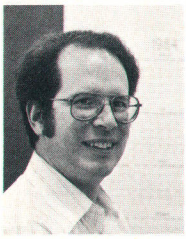


Born in Green Bay, Wisconsin, John Balza studied electrical engineering at the Illinois Institute of Technology (BS 1971) and the University of Wisconsin (MS 1972). He then came to HP and has done code development, production engineering, hardware development, and chip design for several HP computer products. He managed the I/O ROM project for the HP 9835 and HP 9845 Computers and worked on terminal emulators, data communications, and LAN 9000 before taking up his current assignment related to networking for personal computers. John lives in Fort Collins, Colorado, is married, and has two daughters. He enjoys playing both the piano and the stock market.

James L. Willits

An R&D project manager at HP's Colorado Networks Operation, Jim Willits developed data communications products for the HP 3000, HP 250, and HP 9000 Computers. He received a BS degree in mathematics from Kansas State University in 1967 and then served for four years as a computer systems design engineer in the U.S. Air Force, attaining the rank of captain, before resuming his studies at Iowa State University. After receiving an MS degree in computer science in 1973, he joined HP. Jim was born in Sedro Woolley, Washington, is married, has a son and a daughter, and lives in Loveland, Colorado. He enjoys golf, racquetball, downhill skiing, and sailing his Hobie Cat.

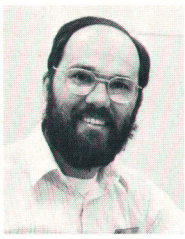
H. Michael Wenzel



Mike Wenzel holds the BSEE (1969) and MSEE (1971) degrees from the University of Denver. His first project after joining HP in 1974 was developing firmware for a raster printer. More recently, he worked on data communications and network software, including design of the message manager and architecture for LAN 9000. He currently is working on new network architecture for the Series 200 and Series 500 Computers. Before coming to HP, he served five years in the U.S. Air Force as a contract officer for the space shuttle program. Mike was born in Alton, Illinois, and now lives in Fort Collins, Colorado. Married and the father of two daughters, he is interested in music, hiking, stained glass, and the use of computers in education (he advises a local grade school regarding networking and computer/software availability to students).

24

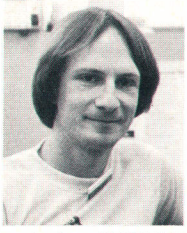
Vincent C. Jones



Vince Jones joined HP in 1979. Now a project manager, his group is responsible for IBM and asynchronous connections to the HP 9000 Computers. Before coming to HP, he specified computer network access and remote sensing systems for the U.S. Air Force and was an occasional consultant to small business computer users. A graduate of Rutgers University (BA and BSEE, 1970), he continued his studies in electrical engineering at the University of Illinois for the MS (1972) and PhD (1975) degrees. He lives in Fort Collins, Colorado, with his wife and three daughters and "enjoys family life in the shadows of the Rockies."

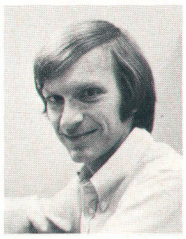
28

Stephen D. Scheid



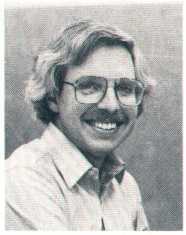
Born in Bryan, Texas, Steve Schied was raised in Phoenix, Arizona, where he attended nearby Arizona State University and received a BS degree in engineering science in 1975 and an MSE degree in electrical engineering in 1978. He then joined HP and worked on QUERY/45—a data base inquiry program, enhanced microcode for the HP 9845 Computer, and most recently, the virtual memory portion of the Series 500 operating system. Outside of work, Steve is a volunteer instructor and trip leader for the Boulder Mountaineering School. He is married to another HP engineer, lives in Fort Collins, Colorado, and has a variety of pets—among them Amazon parrots, which he breeds. An active outdoorsman, he enjoys rock climbing, mountaineering, cross-country and downhill skiing, snowshoeing, and backpacking—last year he backpacked more than 50 miles across Big Bend National Park from east to west.

Dennis D. Georg



Born in Algona, Iowa, Denny Georg studied at Iowa State University, receiving a BS degree in mathematics in 1971 and the MS and PhD degrees in computer science in 1973 and 1975. After teaching computer science for three years, he joined HP in 1978. He worked on HP 9000 software and managed the SUN operating system kernel project before assuming his current responsibility as an R&D section manager. His work on the HP 9000 memory system has resulted in two patent applications. A member of the IEEE, the ACM, and the Planning and Zoning Board of Fort Collins, Denny is married and lives in Fort Collins, Colorado. He enjoys fishing, hiking, amateur radio, skiing, and technical reading.

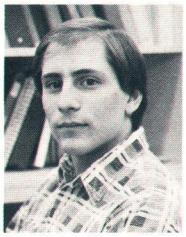
Benjamin D. Osecky



Dan Osecky is project manager for HP 9000 operating system software. Earlier, he contributed to the operating system for the HP 9835 Computer. He is coauthor of a paper on a self-configuring computer network and coinventor for a patent application related to memory management for the HP 9000 Computers. Dan received BSEE (1972) and MSEE (1974) degrees from Virginia Polytechnic Institute and State University, before joining HP in 1976. He was born in Washington, D.C., and is a member of the ACM. Married to another HP engineer, he lives in Fort Collins, Colorado, and is interested in amateur radio, science fiction, hiking, and cross-country skiing.

34

Robert J. Bury



A native of Chicago, Illinois, Bob Bury studied computer science at the University of Illinois (BS 1979). He joined HP in 1980 and helped develop the SUN operating system for the Series 500 Computers. A member of the IEEE, he is married, lives in Fort Collins, Colorado, and enjoys gardening, photography, and cross-country skiing.

38

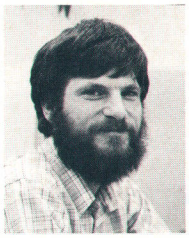
Charles E. Mear, Jr.

Charlie Mear studied computer science at Colorado State University (BS 1977) and the University of Texas (MS 1979) before joining HP. He worked on the file system for the Series 500 Computers and currently is working on the HP-UX kernel for the Series 200 Computers. Born in Midland, Texas, he now lives in Fort Collins, Colorado, is single, and is interested in boardsailing, skiing, and golf.

Marcel E. Meier

Marcel Meier joined HP in 1979 after receiving a BS degree in computer engineering from Case Western Reserve University. He worked on the operating systems for the Series 500 Computers before beginning his current work on HP-UX for the Series 200 Computers. He is a member of the ACM. A citizen of both the U.S.A. and Switzerland, Marcel was born in Manitou Springs, Colorado, and now lives in Fort Collins, Colorado. An avid bicyclist, he enjoys touring and pedals to work year-round. He is also interested in skiing, hiking, sports cars, and audio systems.

Robert M. Lenk



Bob Lenk's contributions have resulted in two papers related to a system for software performance instrumentation. Joining HP in 1981, he worked on the I/O primitives and local area network services for the Series 500. Now he is working on the HP-UX kernel for the Series 200 Computers. A member of the ACM, he holds a BA degree in mathematics (1975) and an MS degree in computer science (1981) from the University of Connecticut. Born in New York, New York, he now lives in Fort Collins, Colorado. He is married and likes square dancing and cross-country skiing.

44

Donald L. Hammond

Don Hammond was recently named director of Hewlett-Packard Laboratories, Bristol, England. He joined HP in 1959 as manager of the quartz crystal department, and in 1963, he became manager of physical research and development. From 1966 to 1979, he was director of the Physical Electronics Laboratory of Hewlett-Packard Laboratories, and from 1979 until his move to England, he was director of the Physical Research Center of HP Laboratories, with responsibility for R&D in medical and analytical instruments, computer peripherals, factory automation, and lithography. He is a member of the American Physical Society, a fellow of the IEEE, and a member of the National Academy of Sciences evaluation committee for the U.S. National Bureau of Standards and the U.S. Naval Observatory. He holds BS, MS and DSc degrees in physics from Colorado State University. A native of Kansas City, Missouri, Don is married and has five children. He served for ten years on the board of trustees of the Palo Alto, California Unified School District, and has been a member of various presidential, gubernatorial, and industry committees on education and technology.

Coping with Prior Invention

by Donald L. Hammond

THIS MONTH, HEWLETT-PACKARD is introducing a new printer, the ThinkJet (HP 2225), which offers what we believe is an unprecedented combination of features: 150 character-per-second printing speed, archival print on ordinary paper, small size, quiet operation, and low cost—both initial cost and total cost of ownership. Power requirements are so low that one model is available with a battery pack that provides more than three hours of printing, or about 200 pages. These advantages have been made possible by a new ink jet printing technology, which we have called thermal ink jet, or more picturesquely, "ThinkJet," to differentiate it clearly from the more common kind of thermal printing, which requires special paper.

We think the story of this technology development is an interesting example of what can happen in today's fast-moving technological environment. In our HP Laboratories at Palo Alto, in the fall of 1978, John Vaught was looking for a new printing method that would have the advantage of inherent simplicity compared with the rather complex electrophotographic process used in the HP 2680A Laser Printer, for which John had designed the optical scanning package.

He started with the idea of turning ink into vapor by high-speed electrolysis and heating, using pressure to eject drops. When this was found to work but with serious failure rates, he conceived the idea of using a small resistor, which when heated for a few microseconds by a current pulse, created bubbles, thereby ejecting drops of ink from a nozzle. This was first demonstrated in March 1979.

We proceeded to develop this idea, amidst some skepticism that the necessary performance and reliability could ever be achieved. The ThinkJet printer is testimony that these concerns were dispersed by extensive development work in several HP organizations on the process and structure. One of the key concepts, originated at HP's Corvallis Division, was a totally disposable ink jet head with a self-contained ink supply.

It is not uncommon, when an important problem such as quality printing receives the attention of many people, that independent conception occurs in isolated research centers. Such was the case with ThinkJet. In September 1981 we learned of the existence of the same concept under development at Canon, Inc., in Japan. Ichiro Endo had conceived the idea independently, with an earlier invention date. Canon referred to the technology as "Bubblejet."

Since we in HP were convinced that this new technology had great promise, the arrival of a new player in this arena caused some concern as to our respective technical positions. There were a number of options but the most attractive for HP was to work with Canon. Excellent ties between the two companies had already been established as a result of our acquisition from them of technology for electrophotographic printers several years earlier.

Hewlett-Packard and Canon have agreed to cooperate in the technology development. Because this process started in 1983, the sharing of technical data has had no major impact on our first product release, but we can feel the positive effect that it is having on our continuing developments. Canon has reflected to us similar feelings. Working with a group that represents a combination of cooperation and competition has provided a valuable perspective, especially increased objectivity, for the technical and management teams of both companies.

This experience has reinforced the principle that technology alone can rarely make a significant contribution in this complex, fast-moving world. There are equally valuable elements, sometimes involving the resolution of relationships in the spirit of international competition and cooperation, that can have dramatic effects on our ability to bring that technology to the market.

We will be reporting in a future issue on more details of these developments, including the ThinkJet printer.

Hewlett-Packard Company, 3000 Hanover Street, Palo Alto, California 94304

Bulk Rate
U.S. Postage
Paid
Hewlett-Packard
Company

HEWLETT-PACKARD JOURNAL
MARCH 1984 Volume 35 • Number 3
Technical Information from the Laboratories of
Hewlett-Packard Company
Hewlett-Packard Company, 3000 Hanover Street
Palo Alto, California 94304 U.S.A.
Hewlett-Packard Central Mailing Department
Van Heuven Goedhartlaan 121
1181 KK Amstelveen, The Netherlands
Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan
Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

CHANGE OF ADDRESS: To change your address or delete your name from our mailing list please send us your old address label. Send changes to Hewlett-Packard Journal, 3000 Hanover Street, Palo Alto, California 94304 U.S.A. Allow 60 days.

5953-8521