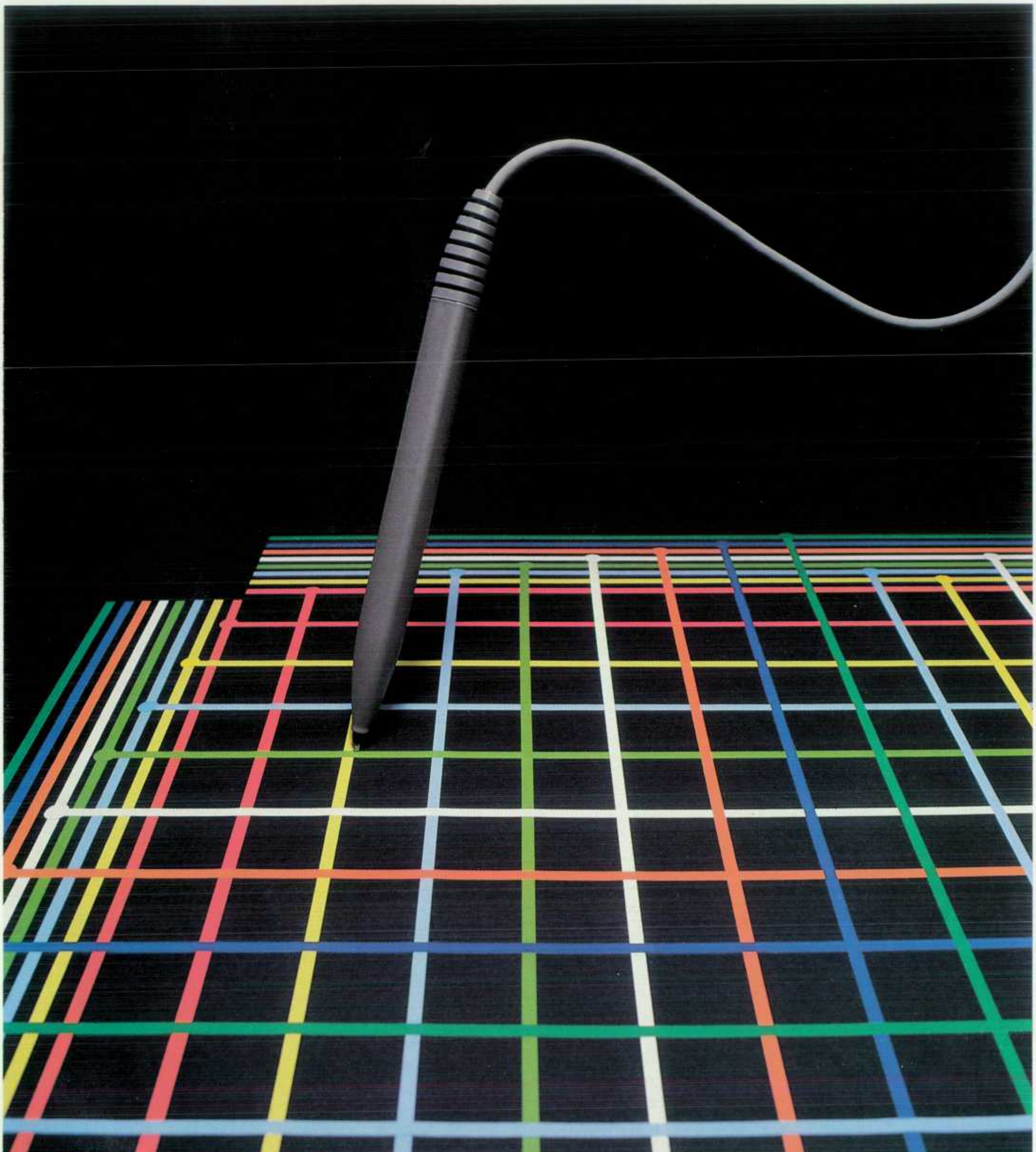
HEWLETT·PACKARD
JOURNAL

JUNE 1987



HEWLETT·PACKARD JOURNAL

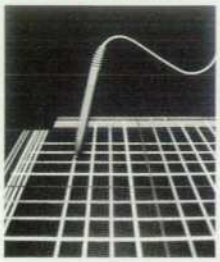
June 1987 Volume 38 • Number 6

Articles

Departments

Editor, Richard P. Dolan • Associate Editor, Business Manager, Kenneth A. Shaw • Assistant Editor, Nancy R. Teater • Art Director, Photographer, Arvid A. Danielson
Support Supervisor, Susan E. Wright • Administrative Services, Typography, Anne S. LoPresti • European Production Supervisor, Michael Zandwijken

In this Issue



Our cover story this month is one of those classic engineering tales. Once upon a time, two HP engineers were discussing how nice it would be to have graphics tablets for their home computers. Wasn't it too bad that the tablets—even HP's—were so expensive? In their free time, they began to design graphics tablets, and they eventually hit on a simple, elegant scheme that reduced the number of parts needed, and therefore the cost, without reducing resolution. The idea is to use just a few conductive traces under the tablet surface instead of hundreds, but to use them over and over in different orders at different locations. They call it permuted trace ordering. Our cover photo illustrates it using a different color for each trace. The graphics tablet, the HP 45911A, costs less than a quarter of what previous tablets cost for the same resolution. The full story is told by Tom Malzbender in the article on page 4.

A graphics tablet is a device that a human uses to communicate graphical data to a computer by pointing with a stylus. You can use it for sketching, drawing, computer-aided design, or menu picking. On the other hand, you might choose some other device, such as a mouse, a touchscreen, the keyboard, a digitizer, or a knob. All of these computer input devices, and others too, operate at human speed, which by computer standards is pretty slow. Hewlett-Packard has a low-cost standard interface for connecting devices of this kind to personal computers and workstations. Called the HP Human Interface Link, or HP-HIL (not to be confused with the HP Interface Bus, HP-IB, or the HP Interface Loop, HP-IL), it allows you to connect up to seven devices to a single port on the computer. To find out how it works, read the article on page 8.

If you're interested in AT&T's UNIX® operating system or in HP's version of it, HP-UX, you've probably already read a lot of the extensive literature on the subject. Even so, you may find some new insights in the paper on page 26, which compares the use and performance of the various interprocess communication facilities available in this multiprocessing operating system. Signals, pipes, shared memory, semaphores, and message queues are ranked for various uses and data is presented to support the ranking.

Branches are decision points in computer programs. Branch analysis is a method of assessing the thoroughness of software testing by keeping track of how many branches have been executed by the test procedure and how many have not. Although it sounds simple enough, the first HP software laboratories that imposed branch coverage requirements on their testing projects found that there are many pitfalls, such as attempting to meet the coverage goal by testing all the easy branches instead of the critical ones. In the paper on page 13, three HP software engineers warn of the pitfalls and lay out a comprehensive methodology for avoiding them and reaping all the benefits of branch analysis.

Yoshio Nishi gained notoriety as the developer of the first commercial 1M-byte dynamic read/write memory chip. That work was done when he headed Toshiba Semiconductor Group's semiconductor device engineering laboratory. Brought to HP by an exchange program between the two companies, Dr. Nishi now directs HP Laboratories' silicon VLSI research laboratory. On page 24, he gives us his view of the current status of CMOS technology and lists some of the engineering challenges facing this technology as we approach the era of ultra-large-scale integration (ULSI).

-R. P. Dolan

What's Ahead

The July issue tells the design story of two instruments for evaluating digital radio performance: the HP 3708A Noise and Interference Test Set and the HP 3709A Constellation Display.

The HP Journal encourages technical discussion of the topics presented in recent articles and will publish letters expected to be of interest to our readers. Letters must be brief and are subject to editing. Letters should be addressed to: Editor, Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304, U.S.A.

Permuted Trace Ordering Allows Low-Cost, High-Resolution Graphics Input

A scheme that substantially reduces the number of trace drivers required provides an inexpensive, but high-performance graphics tablet for HP's HP-HIL family.

by Thomas Malzbender

THE TASK OF ANY GRAPHICS TABLET is to provide the host computer with information corresponding to the position of a pen-like stylus relative to the top surface of the tablet, commonly referred to as the platen. This capability allows the user to input graphical data in a more natural manner for applications such as menu picking, CAD (computer-aided design), sketching, and drawing.

Based on a new input technology, the HP 45911A Graphics Tablet (Fig. 1) represents a significant contribution in price/performance for this class of graphics input devices. Less than a quarter of the cost of earlier HP graphics tablets, the HP 45911A offers a resolution of 1200 lines per inch (0.02 mm) with essentially no jitter at this high resolution. Its active area was chosen to be 11 inches per side to accommodate standard overlays produced by third-party software vendors. Ergonomically, the HP 45911A features a minimal footprint, low-profile package designed to be used in front of large workstations like HP's Vectra Computer without restricting easy access to the system's disc drives. In addition, its standard width of 325 mm allows it to be stacked on top of the system when not in use.

The development history of the HP 45911A is reminiscent of HP's early development style in which projects were initiated by lab engineers with a need for a new product and who believed they had a good idea on how to construct it. Early in 1984, Mike Berke and I were griping about the high price of the HP 9111A (HP's only tablet back then) and how useful a good inexpensive tablet would be for our home computer systems. So when time would allow it, we started experimenting with various tablet designs. After prototyping several technologies (electrostatic, magnetic, optical, and ultrasonic), it became clear that an electrostatic approach was our only choice. Magnetic technology also promised high resolution, but required higher current consumption and had a significant problem with sensitivity being highly dependent on the angle of the pen to the tablet surface.

In an electrostatic design, traces underneath the active

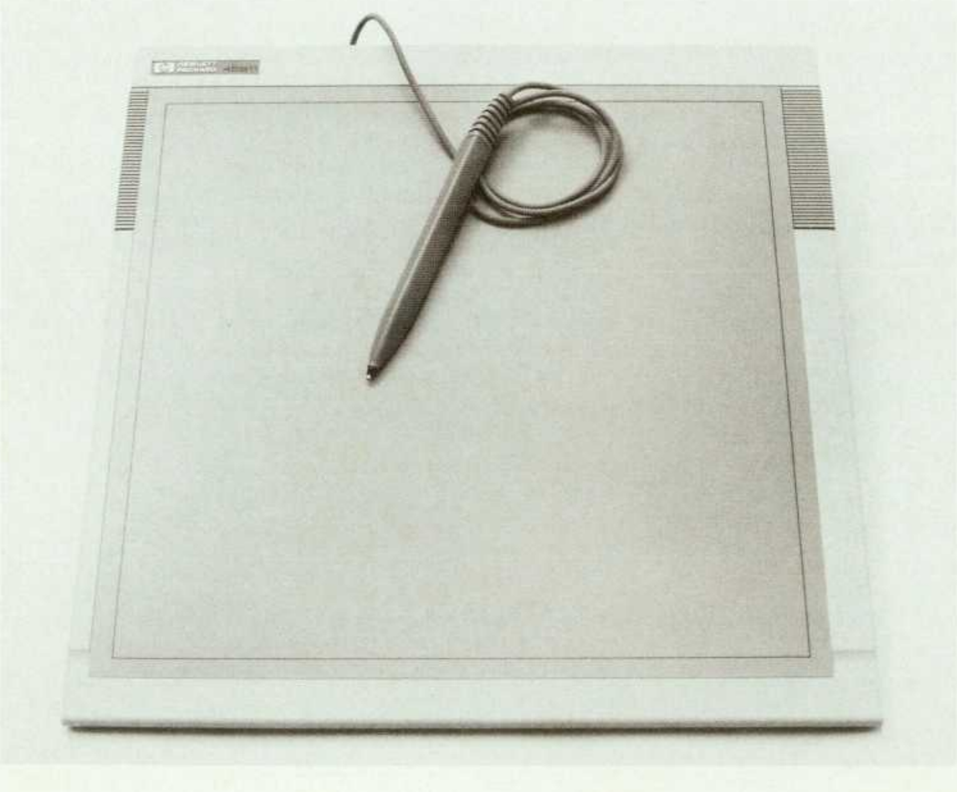


Fig. 1. *The HP 45911A Graphics Tablet features a high resolution of 1200 lines per inch over an 11 × 11-inch active area for a low price. A member of HP's Human Interface Link (HP-HIL) family of input devices, it can be used with a variety of HP computers and workstations.*

area are sequentially pulsed and these pulses are capacitively coupled to the tip of a stylus. The amount of coupling is a function of the local dielectric coefficients (which are normally constant) and the spatial separation between the stylus and any specific trace. Hence, the stylus position can be accurately determined by the relative strength of the signals coupled back from the traces as they are pulsed.

The HP 45911A offers greatly reduced hardware complexity over comparable tablets by using a technique (patent applied for) that reduces the number of trace drive lines from over 110 down to 16. This scheme, called permuted trace ordering (PTO), allows us to drive the traces directly from the on-board microprocessor, eliminating the need for any separate driver ICs, which usually represent a large fraction of the cost of a graphics tablet. To accomplish this reduction, the same drive lines are used over and over again on the 112 vertical and horizontal traces on the tablet by varying the sequential ordering of the traces along the tablet surface. In this way, a unique signature is coupled into the stylus at all points on the platen. Fig. 2 demonstrates this for a section of the tablet platen board. It shows, for both axes, the drivers associated with each of the traces shown. The tablet operates by activating the trace drivers singly, in sequence, and reading the stylus response for each trace driver.

For example, if the stylus is located as shown, the outcome might resemble the list of values shown in Table I. The units in the response column are merely relative values and could be viewed as results from an 8-bit analog-to-digital (A-to-D) conversion.

Table I
X-axis trace driver responses

Driver	Response
0	160
1	60
2	30
3	225
4	100
5	25
6	35
7	50

Observe how the stylus response is a function of stylus-to-trace distance. Driver 3 is the closest and gives the highest response, followed by driver 0, then driver 4. These top three responses can be formed into a code, say 304, which, by design, is unique to that coarse position on the tablet surface. The magnified section of Fig. 2 shows coarse position codes for both X and Y in a small region of the tablet. Note that two different code values occur within a trace spacing. Each of the drivers is pulsed one at a time, the

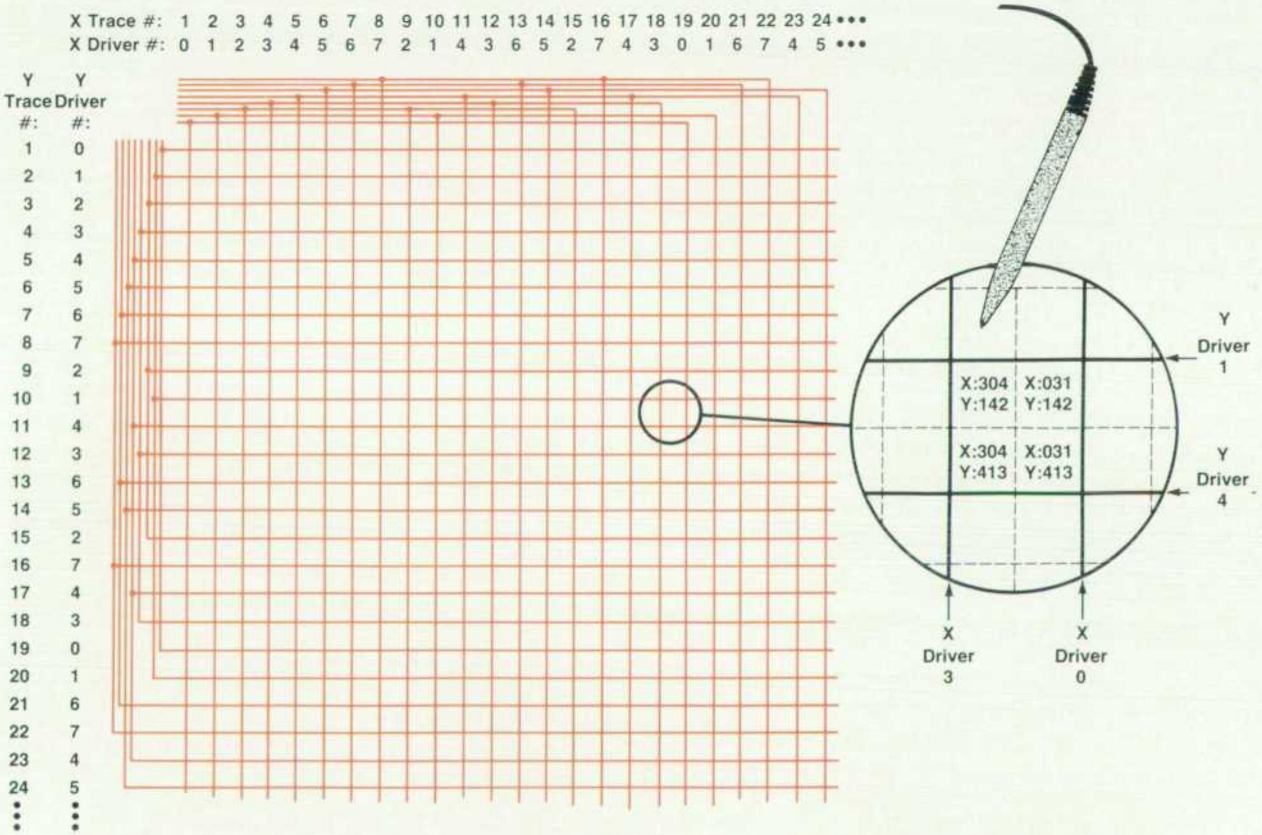


Fig. 2. *A section (left) of the HP 45911A platen with an expanded view (right) showing coarse position codes for X and Y coordinates. The first digit of the code corresponds to the closest trace driver, the second digit corresponds to the second closest, and the third digit to the third closest. The driver sequence is chosen to assure unique codes for all coarse positions.*

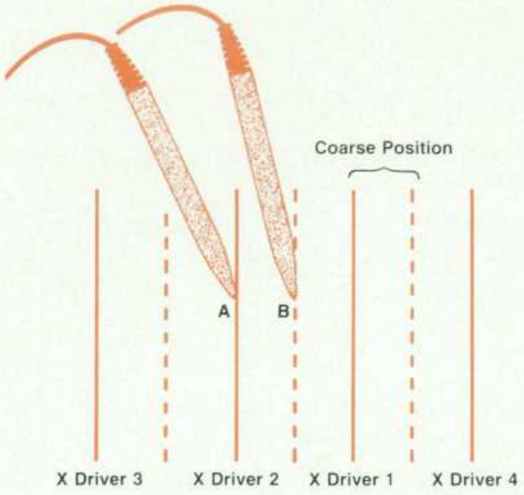


Fig. 3. *At stylus position A, the fine offset value is 0 and at position B the value is 1. Continuous values between 0 and 1 exist between positions A and B.*

responses are measured and sorted, and then the driver numbers for the three strongest responses are combined into a code word. This code word then becomes an address for accessing a coarse-position lookup table.

These algorithms determine coarse position with a resolution of 2.7 mm. To achieve a resolution of 0.02 mm (1200 lines per inch), each coarse position must be resolved into 128 distinct regions. The fine-position routines that accomplish this are based on the equation:

$$\text{Fine Offset} = (V_2 - V_3)/(V_1 - V_3)$$

where V_1 is the magnitude of the strongest response, V_2 is the secondary response, and V_3 is the tertiary response. This relationship was chosen because it represents a computationally minimal relationship with well-defined boundary conditions.

Fig. 3 demonstrates the boundary conditions between coarse position blocks. This fine offset approaches 1 when the stylus is exactly between two traces since the highest response V_1 will have roughly the same magnitude as the second highest response V_2. At the other extreme, V_2 becomes equal to V_3 when the stylus is directly over a trace, since this configuration will yield equal spacing to the adjacent second and third traces. In this condition, the numerator and the fine offset itself approach zero. Between these two extremes, the values are continuous but not necessarily linear. Linearization is achieved through the use of a lookup table within the HP 45911A's microprocessor, and we are left with a flat position response at high resolution.

There is a fundamental relationship between computation speed and noise/jitter performance. Fast position determinations make it possible to use averaging to reduce any noise in the system. For this reason, the fine offset equation is computed in hardware rather than firmware. Referring to the configuration shown in Fig. 4, the subtractor stage is used to generate both the terms $V_1 - V_3$ and $V_2 - V_3$. The first term is applied to the reference input of the analog-to-digital converter (ADC) and the other term is applied to the ADC's signal input. The effect of this is a division of the two terms. Since the speed of this process is limited only by the signal propagation and A-to-D conversion times (dominant here), data can be collected quickly and averaged often. In addition, multiple samples can be taken on the input sample-and-hold circuits, which causes very quick analog averaging to take place there. The result is excellent noise performance.

Noise performance is further improved by two firmware routines, dynamic averaging and antijitter. Dynamic averaging is a technique that offers all the benefits of large amounts of position determination averaging without the drawbacks. Averaging reduces the amount of noise (inherent with the large amounts of amplification necessary to process the minute stylus signal) by the square root of the number of averages. However, conventional averaging causes a perceivable lag when the user moves the stylus rapidly. To overcome this, the dynamic averaging routines change the amount of averaging performed as a function of stylus tracking speed. When the user is moving the stylus quickly over the platen surface, little or no averaging is done to ensure a quick response. With slow stylus movements, large amounts of averaging are performed, which provides excellent noise performance when it is most needed.

Dynamic averaging successfully reduces any jitter down to a single pixel, but no further since the stylus can always sit on the boundary between two pixels. To eliminate this last amount of jitter, changes of only one pixel are not reported.

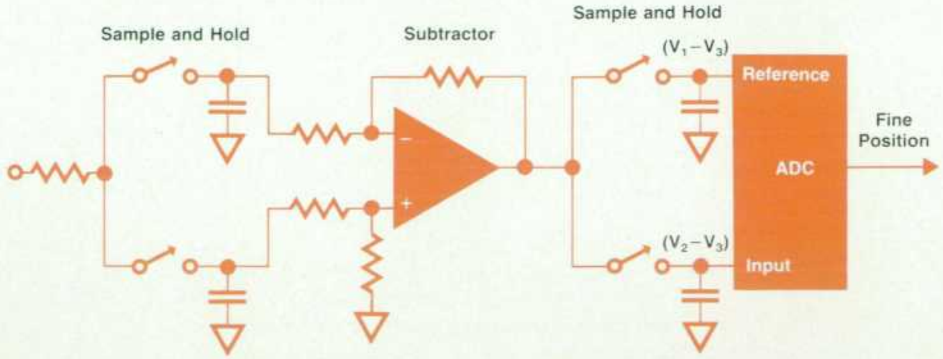


Fig. 4. *Hardware system for computing fine offset position.*

Stylus Design

The conventional electrostatic graphics tablet stylus can be thought of as merely a shielded wire that brings the capacitively coupled trace signals back to the main system electronics for amplification and processing. The stylus for the HP 45911A, on the other hand, is active and the trace signals are amplified at the stylus tip before they are sent back to the main electronic system.

The use of surface mount components (see Fig. 5) let us put the first stages of amplification in the stylus. Although this approach requires power and ground wires to be connected to the stylus, it improves noise performance by roughly an order of magnitude.

Signals seen by the tip are greatly reduced by a parasitic voltage divider formed by any existing tip-to-ground capacitance. In a conventional stylus, the tip and attached wiring running through both the body of the stylus and the cable shield form a considerable parasitic divider.

That is (Fig. 6, left):

$$V_{Stylus} = \left[\frac{C_{Trace}}{C_{Trace} + C_{Body} + C_{Shield}} \right] V_{Platen}$$

Given typical values of 1 pF for C_{Trace} and 100 pF for $C_{Body} + C_{Shield}$, the stylus voltage is approximately $(1/101)V_{Platen}$.

The HP 45911A stylus tip sees only the parasitic tip-to-body capacitance, yielding a signal about ten times stronger at the input to the first stage of amplification.

That is (Fig. 6, right):

$$V_{Stylus} = \left[\frac{C_{Trace}}{C_{Body} + C_{Trace}} \right] V_{Platen} \times \text{Amplifier Gain}$$

In this case, $C_{Body} = 10$ pF and the stylus voltage is approximately $(1/11)$ V_{Platen} multiplied by the amplifier gain.

After buffering by the low-output-impedance amplifier, any shield or body capacitance has no effect. In addition, since the signals entering the first stage of amplification are stronger, the noise level introduced by this stage has less effect, which yields a greatly improved signal-to-noise ratio.

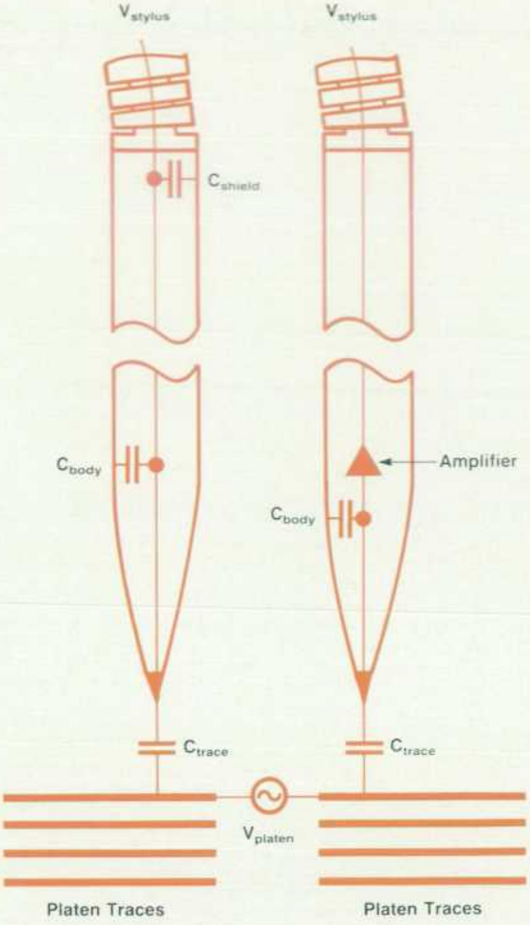


Fig. 6. *Comparison of conventional electrostatic stylus design (left) with HP 45911A stylus design (right).*

Acknowledgments

Mike Berke's technical expertise, insight, energy, and encouragement were involved in nearly all of the functional details of the tablet design. Brainstorming with Mike is what caused the project to happen in the first place. A supportive management environment created by Mark Della Bona and Lorenzo Dunn allowed the project to become a reality. Peter Guckenheimer made contributions early in the development and is essentially responsible for the industrial and mechanical aspects of the stylus. Tom Neal can be congratulated on the industrial design of the tablet itself, and Jun Kato and Dick Bergquam executed a tricky mechanical design. The project was transferred to Singapore in its later phases and Han Tian Phua, Yeow Seng Then, Hock Sin Yeoh, Danny Ng, and others there have made and are still making valuable contributions to the HP 45911A. Also, Rob Starr needs to be thanked for his electrical support near the end of the project.

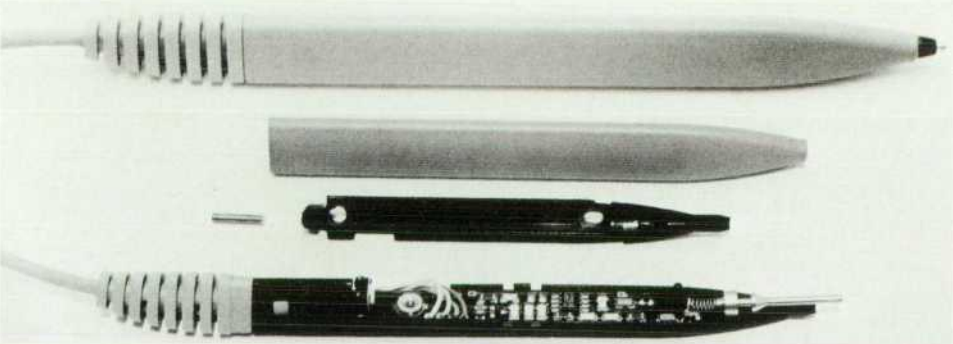


Fig. 5. *Assembled (top) and disassembled (bottom) stylus assembly showing amplifier board using surface mount components.*

The Hewlett-Packard Human Interface Link

Connecting human-input devices to personal computers and workstations is simplified by the definition of an interface link that adapts to the devices on the link and allows them to be added or disconnected during operation.

by Robert R. Starr

THE HEWLETT-PACKARD Human Interface Link (HP-HIL) is an intelligent, low-cost interface for connecting human-speed input devices (e.g., keyboards, mice, and digitizing tablets) to personal computers and workstations. HP-HIL can support up to seven such devices at one time by daisy-chaining them together through a single port on the computer. There are no restrictions on the type and order of the devices connected. Users can easily expand their system by simply plugging in additional input devices.

HP-HIL has become the standard input device interface for HP's personal computers but should not be confused with other types of interfaces such as the Hewlett-Packard Interface Bus (HP-IB, IEEE 488/IEC 625) and Hewlett-Packard Interface Loop (HP-IL),[1] which have distinct and different applications. HP-HIL was designed as an efficient, low-cost method of data collection from human-operated input devices.

Features of HP-HIL

Many PC users find that they need a variety of input devices to handle different data input needs. For example, a mouse is good for many applications, but sometimes the greater precision of a graphics tablet is necessary. HP-HIL allows input devices to be intermixed easily and changed by the user. The HP-HIL protocol identifies and configures devices connected to the computer. This frees the user from the need to change switch settings or configuration menus whenever a device is removed or added to the link. Since HP-HIL will support up to seven devices through a single port, the user does not need a separate interface card for each input device. This can save valuable accessory slots. The input devices receive their power from the computer, thereby eliminating power cords, simplifying the input devices, and lowering costs.

Physical Connection

HP-HIL devices are connected to a personal computer or workstation and to each other in a daisy-chain link. The first device is connected directly to the computer's HP-HIL port. The second device connects to the first device. Each additional device connects to the previous (upstream) device. Up to seven devices can be chained or linked together in this way.

Each input device has two female connectors or ports (except in special cases) while the computer has a single female connector or port. The input devices are connected together by removable cables (usually coiled) having a male plug on each end. Each end of an HP-HIL cable has a differently keyed connector to assure correct connection between the computer and a device or between consecutive devices. The cables and device connectors are marked with one dot or two dots for polarity identification. One dot indicates the upstream connector of a device or the downstream end of a cable. Two dots indicate the downstream connector of a device or the upstream end of a cable. An example of how devices can be interconnected is shown in Fig. 1. Some input devices have only one HP-HIL port because of size limitations. An example is the HP 46060A Mouse. Devices having a single port must be the last device on the link.

HP-HIL Architecture

The HP-HIL architecture is an extendable serial interface consisting of a personal computer or workstation (master) and from one to seven input devices (slaves). The master provides power, ground, data-out, and data-in signals to the devices through a shielded four-conductor cable. The master contains an integrated circuit, the master link controller (MLC), that provides the hardware interface between the system processor and the devices connected to the link as shown in Fig. 2. Similarly, each device contains an IC, the slave link controller (SLC), that provides the hardware interface between the link and the input device's microcontroller as shown in Fig. 3.

In the PC, the MLC functions much the same as a UART

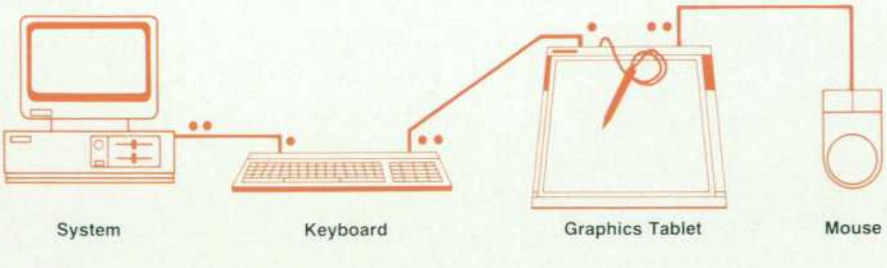


Fig. 1. *Typical HP-HIL device interconnection. The dots indicate upstream and downstream polarity as shown and are marked on the device connectors and cable ends.*

(universal asynchronous receiver-transmitter). The MLC accepts commands directly from the master's processor over a bidirectional eight-bit data bus and transmits the messages (called frames) in serial form onto the link in the proper format. The MLC also accepts serial data from the link and places the data in its 16-frame FIFO (first-in, first-out) buffer for retrieval by the master's processor. The FIFO buffer queues incoming frames to reduce the number of interruptions to the host processor. Error checking and loopback modes for local testing are also supported by the MLC. The MLC requires an 8-MHz clock, which can be supplied by the master's processor clock or generated locally by adding a ceramic resonator to the MLC. Two interrupt lines, nonmaskable and maskable, are available from the MLC. The nonmaskable interrupt can be used for hard resets to the master's processor generated by keyboards (e.g., ctrl-shift-reset keys depressed to generate a system hard reset). The two signal lines, SO and SI, are protected against electrostatic discharge (ESD) damage by clamp diodes between the +5V supply and ground, and by resistors in series with the signal lines.

Each HP-HIL device contains the SLC and a microcontroller. The SLC provides the interface between the link controller and the device's microcontroller. The SLC receives commands from the device microcontroller, transmits data, retransmits commands, and detects communication errors. The SLC also provides self-test capabilities useful during power-up. The clock is provided by the SLC using an external 8-MHz ceramic resonator, and is divided to 4 MHz for the microcontroller's use. Communication with the device processor is serial and is designed for use with National Semiconductor's COPs family or similar microcontrollers. Link protocol is handled by the device microcontroller, which also handles data collection from the input device (key array, optical encoders, etc.). Like the MLC, the signal lines of the SLC (SI, SO, RI, and RO) are protected against ESD damage.

An integral part of HP-HIL is the ability to supply power to the input devices from the personal computer or workstation. 12Vdc is supplied to the link from the master's power supply for use by the input devices. Devices locally regulate the 12Vdc to 5Vdc so that any voltage losses in the cables do not affect the devices. Most input devices require less than 100 mA.

HP-HIL Input Devices

- HP-HIL Touch Accessory, HP 35723A. A 12-inch user-installable touchscreen bezel which provides touch interaction with the host computer. The HP 35723A features a resolution of 43×57 points maximum.
- HP-HIL Graphics Tablet, HP 45911A. An 11×11-inch graphics tablet with 1200-lines-per-inch resolution.
- HP-HIL Keyboards, HP 46021A and HP 46030A. The HP-HIL keyboards are generally supplied with the computer for which they were designed (e.g., HP TouchScreen II, Vectra, and HP 9000 Series 300). These keyboards are available in a wide variety of languages.
- HP-HIL Mouse, HP 46060A. This mouse simplifies the task of positioning the cursor on the screen. It has 200-counts-per-inch resolution and two buttons.
- HP-HIL Rotary Control Knob, HP 46083A. This module provides two-axis relative cursor positioning via a rotary knob and a toggle key. It has a resolution of 480 counts per revolution.
- HP-HIL Security ID Module, HP 46084A. The ID Module allows users to run secured application software. It returns an identification number for identifying the computer user and is used in application programs to control access to program functions, data bases, and networks.
- HP-HIL Control Dial Module, HP 46085A. This module has nine graphics positioning dials. It is used in graphics display applications to provide three-axis rotate, translate, scale, and other attribute functions. Each dial has a resolution of 480 counts per revolution.
- HP-HIL 32-Button Box, HP 46086A. This box provides 32 user-definable buttons for menu selection and one user-programmable LED. It is used in CAD/CAE applications.
- HP-HIL Digitizers, HP 46087A and HP 46088A. These digitizers are for use in interactive graphics, graphics entry, and menu selection applications. Two active area sizes (A or B) are offered. They each have a resolution of 1000 lines per inch. An optional cursor with crosshair is available.
- HP-HIL Quadrature Port, HP 46094A. This product provides a nine-pin subminiature connector for interfacing quadrature signals to HP-HIL. Three keyswitches are supported.
- HP-HIL Bar-Code Reader, HP 92916A. This module provides an alternative to the keyboard for data entry applications. It reads UPC/EAN/JAN, interleaved 2-out-of-5, Codabar (MHI and USD-1), 3-of-9, and extended 3-of-9 codes.

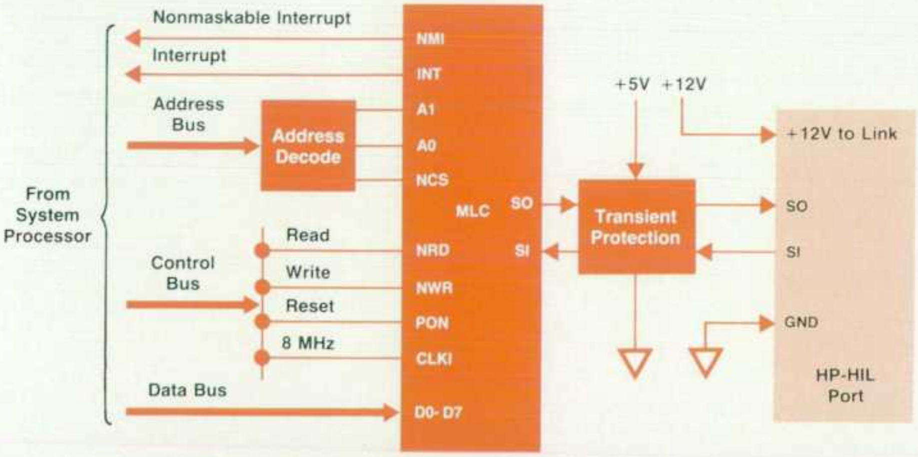


Fig. 2. *Hardware interface between personal computer or workstation (master) processor and HP-HIL.*

Data and Data Paths

Data moves around the link in packets called frames. A frame consists of fifteen bits including one start bit, one stop bit, one data/command bit, three address bits, eight data bits, and one parity bit (see Fig. 4). Frames are transmitted around the link at the rate of ten microseconds per bit, or 150 microseconds per frame. Frames are transmitted at a maximum of one frame per 154 microseconds, or about 6,500 frames per second. Four microseconds are left between frames to avoid collisions. When the link is being used for data collection from input devices, devices are polled (asked if they have data to report) about sixty times each second. This is an acceptable data collection rate in that there is not a perceived delay between user input and the personal computer's response. A maximum of fifteen bytes of data can be collected with each poll.

The idle state of the link is a logic one, with the first bit in a frame (the start bit) at logic zero and the last bit (the stop bit) at logic one. The parity bit is computed so that the total number of logic-one bits in the 15-bit frame (including start, stop, command, parity, address, and data bits) is odd.

Frames can represent either data or commands. The command/data bit indicates whether the eight-bit data field contains data or the opcode of an HP-HIL command. All frames have a 3-bit device address so that commands and data can be associated with a particular device. Command frames generally have a universal address which directs a command to all the devices on the link. When a frame is received by a device, the device always checks for an address match (a universal address or device address matching its own). Frames received that have a matching address or a universal address are acted upon by the device. Frames received that do not have a matching address are retransmitted by the device's SLC.

Command frames always originate from the master, with two exceptions. If a device detects an error (e.g., a frame received by the device is corrupted), then the device originates a command frame indicating an error has occurred. The second exception is when a system hard reset command frame is generated by a keyboard.

When communicating with devices, the master transmits a single command frame or data frame(s) plus a command frame onto the link. HP-HIL protocol allows only a single command frame to exist on the link at any given time. However, multiple data frames can exist on the link. There are five scenarios that can occur when the master transmits on the link.

In the first case the master transmits a single command frame onto the link. The master waits until that command frame passes through all devices and returns before taking further action. The returning command frame is generally identical to the command frame originally transmitted by the master. For some commands, the frame may return to the master modified to indicate some specific information about the link.

The second case is a subset of case one. Here, the master transmits a single command frame onto the link, but the command frame is not expected to return. The master waits a predetermined time, and then proceeds with further transmissions onto the link. The waiting period is called a "time-out" and occurs when no frames return after a command is transmitted. If a time-out also occurs when a frame is expected to return to the master, then the master interprets the lack of response as an error condition and takes the appropriate action.

In the third case, a single command is transmitted by the master, but data frames are returned before the command frame returns. The number of data frames returned depends upon the command transmitted, and whether the device(s) have data to return to the master. Some commands collect data from a specific device while other commands collect data from several devices. Up to 15 data frames can be returned in response to a single command. As always, the data frames contain the device address so the master can identify the originator of each data frame.

In the fourth and fifth cases, the master transmits one or more data frames and a command frame. In the fourth case, a single data frame is transmitted followed by a command frame. The data frame contains register address information

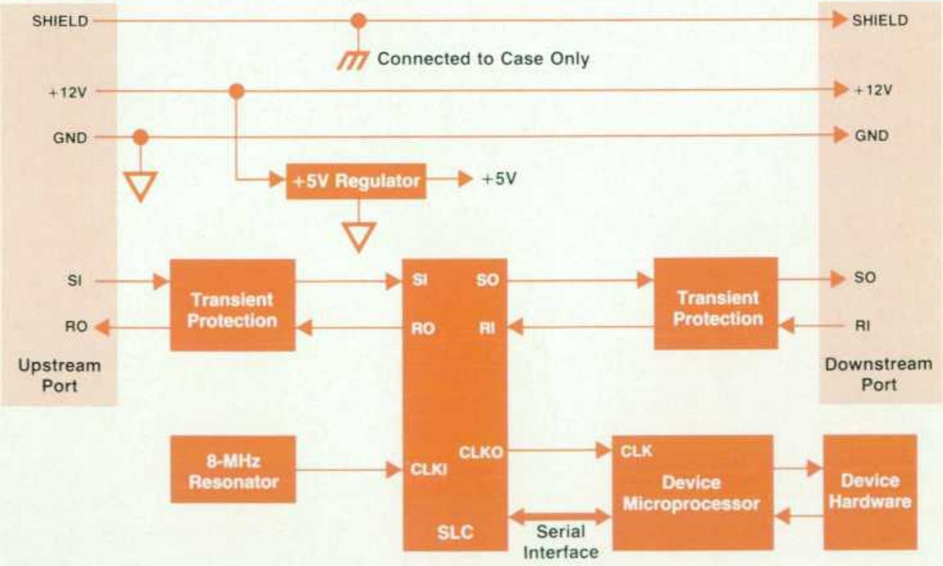


Fig. 3. *Hardware interface between HP-HIL and input device's microcontroller.*

for a particular device. The device responds by transmitting the contents of the addressed register followed by the original command frame. The master waits until the data frame and command frame are received before transmitting additional commands onto the link. The fifth case is similar to case four, but here the master transmits several data frames followed by a command frame. The addressed device returns only the command frame back to the master in case five.

Consider a system with two devices attached as shown in Fig. 5. When the master's MLC transmits a frame, it does it on the serial data-out (SO) line. The frame is received on the serial data-in (SI) line of the first device's SLC. The first device checks for an address match. If no match is found, the frame is retransmitted on the first device's SO line. Assume that the transmitted frame is a command with an address for device one. Device one finds an address match and then proceeds to act on the command. After the command is processed, device one retransmits the data frames (if any) followed by the command on its SO line. Device two then receives the frames on its SI line and checks for an address match. Any data frames originated from device one would have device one's address, and thus would have no address match with device two. The command frame also has device one's address and thus no address match occurs there either. The frames are then retransmitted on device two's return data-out (RO) line rather than on its SO line. This is because device two is the last device on the link for this example. The last device is set to return data on RO rather than SO during the configuration process that occurs upon link startup. The frames are received on the return data-in (RI) line of device one which then passes the frames directly (buffered only) out on its RO line back to the master. The master receives the frames on its SI line to complete the process. This entire process takes about 8 ms or less, depending on the number of devices connected and the amount of data returned.

HP-HIL Protocol

Automatic polling (data collection from the devices) is also possible by the MLC and requires processor intervention only when data is received. The master's processor can also be hard reset by the HP-HIL devices through the MLC.

Each input device on the link is assigned a unique address so that devices can be distinguished from one another. Frames have three address bits, which allows for eight unique addresses. Addresses one through seven are used for devices on the link. Address zero is reserved as a universal address which is used when a command is to be acted upon by every device.

HP-HIL has a command set through which all necessary functions to set up and maintain the link are performed. The commands can be grouped into five categories: configuration, error recovery, data retrieval, identification, and special functions.

Configuration is the process by which the link is set up so that the master can collect data from input devices in an orderly manner. Since the master does not know what devices and how many devices are connected to the link upon power-up, the configuration process must occur before the link can be used. Configuration typically occurs when the master is first powered up. The configuration process is handled by the master's firmware and requires no user intervention. The goal of the configuration process is to assign a unique address to each device on the link, and to set the device modes so that data will be looped back by the last device on the link. (The SLC in a device can internally loop data back to upstream devices or pass data on to downstream devices.) In the process, each device is requested to identify itself (report an ID code) so that the master will have the necessary parameters for scaling the device data. Also, the master can determine if any device supports advanced features.

HP-HIL provides several levels of error recovery. If an error occurs, the error recovery process will preserve the maximum amount of data and minimize the master's interaction with the link. Although recovery is performed by the master's firmware, errors can be detected either by devices or by the master. For example, an error might occur when a user disconnects a device while it is reporting data. The disrupted frame(s) would be detected by the master, causing error recovery to begin. Although a part of error recovery, disconnecting devices is considered a normal part of link operation. If data is lost from devices still remaining on the link, the data can be recovered. Devices save the data that was last transmitted so that if an error occurs, the master can request the data again.

Data retrieval or polling is the process by which the master gathers information from the input devices connected to the link. Keyswitch transition data, character data, position data, and a limited amount of status informa-

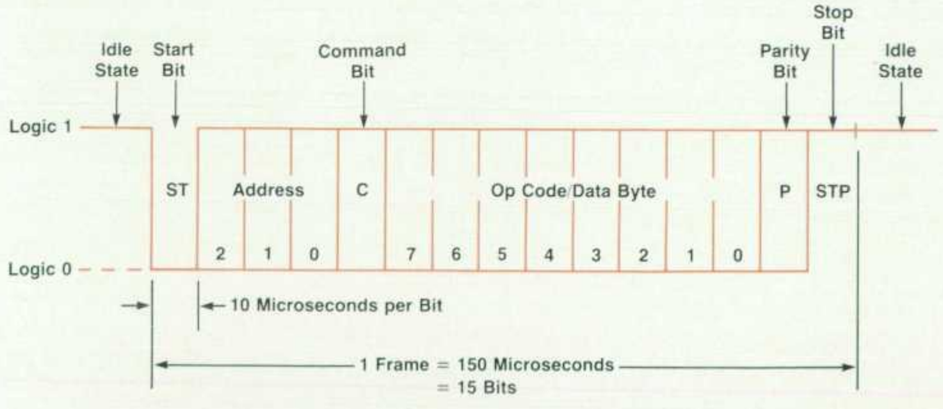


Fig. 4. *HP-HIL frame format.*

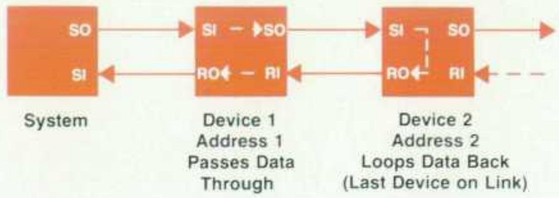


Fig. 5. *Block diagram of system with two input devices attached via the HP-HIL.*

tion can be communicated back to the master using the data retrieval commands. When used with information obtained by the identification commands, input device data can be completely processed. The master can request data from a specific device or all devices. A maximum of 15 bytes of data can be returned in response to a single poll command. Devices determine if there is sufficient room to add their data to a passing poll command. Since the command always trails the data, the device inserts its data and then retransmits the command. Because there is always an address associated with the data, the master can easily determine where the data originated.

Identification commands are used to determine the type of the attached devices and their characteristics. Device types could be keyboards, relative positioning devices (e.g., mice), absolute positioning devices (e.g., digitizers), or other devices. Characteristics of devices are typically resolutions (counts per centimeter), maximum counts, directional information, and information on how the device reports data. Devices also carry an internal identification byte called the device ID byte. This ID byte is assigned to the device during its development and is placed in the device's microcontroller code. The ID byte is used to identify the type of device (keyboard, relative positioning, etc.) and nationality (native language) in the case of keyboards or keypads.

HP-HIL can be used in other ways than efficient collection of data from input devices. Special functions include commands designed to take advantage of advanced features supported by some input devices. These features are related to register reads/writes, keyswitch autorepeat, output, and system reset. These capabilities are not required for basic HP-HIL operation, but they are available for devices with special requirements. For higher-speed data transfers (up to 6,500 bytes per second), register-oriented commands from this category would be used.

System and Device Control

Controlling the link through the MLC consists of four basic activities: configuration, error recovery, polling (data collection), and adding devices to the link. Upon system startup (usually part of the power-on routines), the master's processor performs a self-test on the MLC. Then the configuration process begins. The processor issues the appropriate commands to identify and configure any devices on the link. Once the link is configured, polling begins and is the main activity of the MLC. Occasionally (perhaps once a second), the processor will have the MLC issue commands that will detect if a device has been added to the link. If so, the new device is configured into the link and polling resumes.

Device control consists of three activities: initialization, servicing interrupts, and collecting data. Initialization occurs at power-up and causes the SLC to perform a self-test. The device microcontroller also performs a self-test. When the self-tests are complete, the microcontroller places the device's SLC in its appropriate power-up mode, concluding the initialization process. The microcontroller then spends the rest of the time checking for interrupts from the SLC and collecting new data. An interrupt is generated by the SLC whenever a frame is received. The device's microcontroller then begins an interrupt service routine which may be simple or involved depending on the frame contents. When not servicing interrupts, the microcontroller looks for any new data from the device's input mechanisms. This could be sensing a key depressed, or checking for a transition on an optical receiver, etc. When data is available, the microcontroller formats the data so that it will be ready for transmission to the master.

Acknowledgments

The author would like to thank the key contributors to HP-HIL: Mark Brown for his efforts in hardware and protocol development, Carol Bassett who designed the MLC and SLC, and Greg Woods for his contributions to the HP-HIL protocol.

Reference

1. R.D. Quick and S.L. Harper, "HP-IL: A Low-Cost Digital Interface for Portable Applications," *Hewlett-Packard Journal*, Vol. 34, no. 1, January 1983, pp. 3-10.

Software Verification Using Branch Analysis

Imposing branch coverage requirements on a software testing project can be counterproductive unless a comprehensive branch analysis methodology is followed.

by Daniel E. Herington, Paul A. Nichols, and Roger D. Lipp

BRANCH ANALYSIS IS A METHOD of assessing the thoroughness of software testing. The method consists of inserting procedure calls, called probes, into the code at all of the decision points. These probes make it possible to monitor the execution of specific portions of the software.

This paper addresses the problems and issues of using branch analysis during software testing. We begin by discussing common software testing metrics, including the branch coverage metric. We then discuss software testing, both functional and structural, using branch analysis.

When we first began using branch analysis as a requirement for the release of systems software, many problems occurred because we started without a clear understanding of this coverage metric and there was no known testing methodology using this metric. The final sections of this paper discuss what we have learned from the use of branch analysis and the testing methodology we have developed to provide for an efficient, cost-effective, and quality-conscious software verification process using branch analysis.

Verifying Software Quality

Software quality can be, and is, measured by a number of different elements, including functionality, usability, reliability, and others. Often, the most visible aspect of software quality is conformance to specifications, or more realistically, nonconformance to specifications—that is, errors. Because software is conceived, specified, designed, and built by humans, there are usually plenty of these nonconformances in any large software system. Software quality can be improved by avoiding these errors using software engineering techniques, or by finding and removing them using software verification techniques—usually both. This paper addresses a special type of software verification.

Software can be verified in two ways: statically, using desk checks, walkthroughs, inspections, static code evaluation tools, and the like, or dynamically, using any of a number of software testing techniques.

For managers to be able to control the software verification process, they need to be able to quantify this process. To this end, a variety of software metrics are collected. The most important of these for the purposes of this paper is branch coverage. Branch coverage is a metric that identifies how much of the software, at the source code level, has been executed by the test suite.

We started collecting and using the branch coverage metric in 1981. At the time, there was no known testing methodology using this metric. As a result, a number of interesting and often frustrating problems occurred. We have been focusing considerable attention on clearly understanding these problems and the solutions that various project teams have proposed and tried.

We have identified a complex set of reasons for the problems. One is that our expectations of testing were raised markedly because we now had detailed information on testing coverage. We also had a difficult time trying to identify and develop tools that would help our engineers test specific sections of code.

We have found some positive process improvements that have helped our engineers conduct branch analysis testing in a much more cost-effective and quality-conscious manner. We have also developed a methodology that combines all of these new techniques to provide a structural software verification process that is free of many of the problems we encountered.

Software Testing Metrics

Thorough and effective testing is paramount to the success of a software product. Proving the software reliable is an important part of the testing. To accomplish this, the software must be fully exercised, and any defects detected along the way must be removed. This is an extremely difficult, tedious, and complex activity, frequently taking 40 to 60 percent of the total project effort. It can also be a time of frustration. Questions such as "How much testing is enough?" and "What are the testing results to date?" are frequently asked. The information needed to answer these questions is often not available.

Many of these problems can be overcome or controlled, at least to some extent, by using good metrics. These help to assess progress, report status, and assist in decision making. No one metric is sufficient; rather, a number of different, well-defined metrics are required. Each provides information about a different aspect of the process, and they combine to form a complete and accurate picture of the testing process.

The metrics of interest to this discussion fall into two general categories. First, there are those that quantify the testing coverage, and second, there are those that quantify the software's reliability.

Testing coverage must be quantified and assessed from both an external, or functional, and an internal, or structural, point of view. For the former, assurance needs to be given that all system functionality has been exercised. Tests to validate external features are identified from the specification. They are then created, and their execution progress is tracked using a function matrix. This matrix provides a quantification of functional testing coverage.

Testing the software from only an external point of view is insufficient. Many conditional branches or decisions made by a piece of software are there to support internally implemented functions. This is especially true in system software. Testing from only an external point of view will rarely exercise all of this logic. An internal view of the software is required. Additional tests can be derived by using the internal software specifications. To assure complete structural coverage, however, the tester must be able to determine not only that the software functions have all been exercised, but also that all software logic has been exercised. Hence the tester must be able to monitor what is being executed inside the software. This can be achieved by using a tool to collect the branch coverage metric. This branch analysis tool tells the tester which branches in the code have been executed and which ones have not. Using this data, additional test cases can be derived to exercise untested software logic.

Quantifying software reliability means tracking problems detected and evaluating product stability. All detected problems must be classified and analyzed. Problems are divided into those affecting reliability and those affecting other aspects of quality, such as ease of use or documentation clarity. Reliability problems, or errors, are analyzed to assess severity, to determine their true causes, and to identify where in the software they were found and how they were discovered. Corrective action to resolve them is also tracked. Software stability is assessed by determining how long between failures the software will run under a defined load. This is typically known in the hardware world as mean time between failures (MTBF), and it is just as important for software as it is for hardware. Progress is determined by observing a decrease in the frequency of the detection of errors and an increase in MTBF.

A complete set of testing metrics must contain those pertaining to coverage and those pertaining to reliability. Coverage metrics include test case matrices and branch analysis. The matrices quantify the testing of external features, while branch analysis quantifies the testing of the internal logic that implements these features. Reliability metrics include discovered defects and mean time between failures. All of this information helps quantify and improve the effectiveness of the testing process. Over the past several years, use of these metrics has been increasing and their value has become clearer and better understood. At the same time, the understanding of how testing metrics affect the testing process has also improved. For instance, we have found that knowing which tests exercise which portions of the logic improves the testers' understanding of the software. We have also found that more defects are likely to occur in a module with high decision (branch) density (see Fig. 1). However, we have also found that using an incomplete set of metrics can have a detrimental effect on the testing process.

The Software Testing Process

The two major classifications of software testing are functional testing and structural testing. Functional testing is also referred to as black-box testing because it is conducted by viewing the software as a black box. In other words, the tests are written with no knowledge of the internal structure of the program. In fact, functional tests are most often written from the software specifications before the code has even been written. Structural testing, or white-box testing, takes the opposite point of view. The major concern of structural testing is to ensure that all of the code of the program has been executed during testing. This is accomplished by monitoring the program's execution during testing.

Functional Testing. The two most widely accepted techniques for functional testing are equivalence class partitioning and boundary value analysis.[1-6]

Equivalence class partitioning is a technique designed to partition the input domain into classes such that if one test case from a class is executed and fails to find an error then any other test case in that class would fail to find an error. This technique is essentially designed to reduce the number of tests necessary to verify that the code meets its specifications.

Boundary value analysis is actually a spin-off of equivalence class partitioning. Since most errors are found at or near the boundaries of the equivalence classes, boundary

Networking Product

Prerelease Branch and Error Density Data

	Avg. BR/KNCSS	KP/KNCSS
24 Components	141.24	2.39
5 Comp. with Avg. BR/KNCSS < 111.24	91.83	2.29
6 Comp. with Avg. BR/KNCSS > 171.24	198.91	3.81

Legend

Avg. BR/KNCSS: Average branches per 1000 noncomment source statements

KP/KNCSS: Known problems per 1000 noncomment source statements

Fig. 1. *Data compiled for the components of a networking software product shows that more defects are likely to occur in a module that has higher branch density.*

value analysis is used to assure thorough testing of boundary conditions. This is done by writing additional test cases immediately before, at, and immediately beyond the boundaries of each of the equivalence classes. This also tests for detection of erroneous inputs beyond legal boundaries.

Structural Testing. A major weakness of functional testing is that there is no way to be sure that testing is complete. Structural testing is used to help alleviate this weakness. Structural testing is conducted using a special, instrumented version of the software which contains transparent procedure calls (called probes) which are inserted into the code by the compiler. When the program is tested, the probes log which sections of code are executed by the tests.

The most obvious benefit of structural testing is the identification of untested code. All of the code in the system was included to provide some type of functionality. Some of this code may be the implementation of internal features that would not be apparent from the specification. Functional test cases are written to test the external functionality. If some of the code goes unexecuted after this testing then part of the implementation of the functionality has not been tested and there is a potential for errors to go undetected. Structural testing can help alleviate this problem by identifying untested code. Typically, the automated functional test suite is run with an instrumented version of the software to evaluate the completeness of the test suite. We have found that for system level software, these automated tests leave 40% to 60% of the branches unexecuted. Some of the missing tests are functional tests that were overlooked during the development of the test suite, but most of them are actually structural tests of functionalities that are transparent to the user-level functionality of the system. These include error recovery processing and housekeeping processing that must be done before the system can do its specialized function. The key point here is that without structural testing these holes in test suites cannot be exposed.

Another benefit of structural testing is the quantification of the testing effort. The first line of Tom DeMarco's book *Controlling Software Projects*[7] is: "You can't control what you can't measure." Structural testing is completely quantifiable, giving us much greater control of the testing process.

Conducting Branch Analysis. Fig. 2 shows the typical flow of activities when conducting branch analysis on a software component. First, functional testing is conducted using the traditional functional testing techniques. These tests are rerun with the instrumented program to measure the test suite. The data is compared with the coverage criteria to see if the current level of coverage is sufficient for release of the system. If the coverage is not sufficient (the normal case), the branch analysis data and the source code are analyzed to identify untested functionalities. New test cases are created to test these functionalities. These tests are implemented and run using the instrumented program. If errors are found by the new test cases, the errors are fixed and the regression package is augmented and rerun to make sure no new errors have been introduced by the fixes. At this time the new coverage number is compared with the release criteria again. This loop is followed iteratively until the release criteria for branch coverage are met.

Branch Analysis Problems

Branch analysis is not the perfect testing aid. Some interesting problems occurred when branch analysis requirements were set on projects throughout the computer sector of HP.

Branch Analysis Doesn't Prove Correctness

The first problem we ran into was a misunderstanding of the branch coverage metric. It is very important to realize at the outset that there is not necessarily any correlation between high branch coverage and low defect rates. The branch coverage metric is a meaningless number if the testing is not conducted properly. High branch coverage alone will not expose errors. Branch analysis has all of the weaknesses of structural testing in general. For instance, there is no way branch analysis can expose missing-code errors. These occur when part of the functionality of the system described in the specification is overlooked and doesn't exist in the code. Since branch analysis just measures what is present in the code, no tests will be shown to be missing from the test suite for the missing code, even though there is functionality in the specification that hasn't been tested. Another weakness of structural testing shows

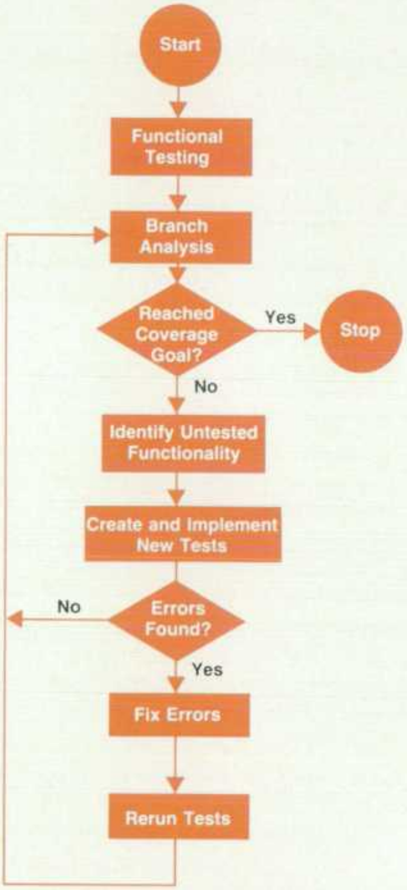


Fig. 2. *Flow chart for a typical software project using branch analysis.*

up in systems software that must run concurrently. Because the branch analysis metric has no conception of timing, code that has timing errors in it may be logged as tested without exposing these errors. In other words, some of the code may exhibit errors only when another specific event occurs at a specific moment. Branch analysis also suffers from many of the weaknesses of software testing in general. This can be illustrated by one of the most noted weaknesses of software testing: coincidental correctness. Consider the following Pascal code:

```
.
.
.
readln(NUM);
X := NUM * 2;
writeln('result = ',X);
.
.
.
```

If the input to the readln is a 2, the output of the writeln would be:

```
result = 4
```

Now suppose the second statement above was supposed to be:

```
X := NUM + 2;
```

or

```
X := NUM ** 2;
```

In all these cases the output would be the same for the input given. However, the program logic may still be wrong. Branch analysis compounds this problem by giving a report that proves that this code has been executed. Since the output was correct, this report gives the tester a false sense of security with respect to this code.

The main point here is that, by itself, neither functional testing nor structural testing is satisfactory. They must be combined in a systematic method to make sure they are both used to their best advantage.

Scheduling Conflicts

It turned out that increasing testing coverage to meet our coverage requirements was considerably more difficult than we had anticipated. Since we were setting these requirements on projects that were already under way, and in some cases the code was already completely integrated, we immediately ran into scheduling conflicts. This put a great deal of pressure on the test engineers to reach the coverage requirements as quickly as possible. This gave them an incentive to raise the coverage using whatever tools and techniques were available. Since there were no proper tools available at the time, and no techniques specifically for this type of testing, our engineers were conducting testing in a relatively *ad hoc* fashion. This caused an intensification of the rest of the problems discussed in this section. In other words, in this case, the concept of learn by doing turned out to be very costly indeed.

There was no real solution to this problem. The hard fact was that we were raising the expectations of our testing and therefore had to extend the expectations of our schedules for that testing commensurately. It should also be noted that the additional testing that was being conducted was also finding more errors. This made it more difficult to keep up with removing them as well.

Increasing Difficulty

Another problem worth noting is caused by the fact that branch coverage becomes increasingly more difficult to improve. This increase in difficulty is nonlinear; the higher the current branch coverage, the more difficult it is to improve the branch coverage (see Fig. 3). The reasons for this are fairly simple. First, we have found that roughly two thirds of the untested branches are in error detection and recovery code. This is understandable, because systems software must protect its own logic from the actions of many other processes. These errors must often be checked at many points in the system. Since the offending processes may or may not be under the control of the software under test, it can be very difficult to cause these errors to occur at the precise time required for particular error checks.

Another reason for increased difficulty is that as the branch coverage increases, the untested branches tend to become widely dispersed in the software. This often means that fairly elaborate tests must be created to test one or two previously unexecuted branches.

Ineffective Testing Tendencies

The results of the problems described above, compounded by the lack of proper tool support, are several tendencies that represent attempts to circumvent the problems, but have the effect of cheating the testing process.

Testing Easy Branches. The first tendency is to test all of the easy branches instead of the branches that really should be tested. An easy branch is a branch that requires very little effort to execute. Unfortunately, the easy branches are not necessarily where the errors are! Hence, if testing

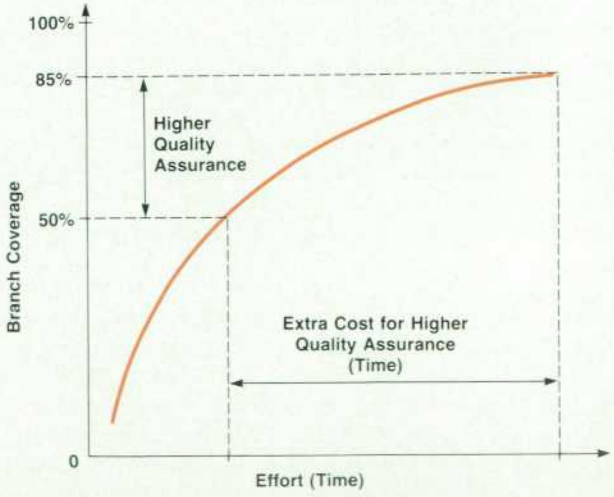


Fig. 3. *When branch coverage is higher, more effort is required to improve it.*

is done with the sole intention of testing as many of the easy branches as possible, leaving the more difficult branches unexecuted, then the value of the testing process has been degraded.

Ignoring Possible Decision Errors. Another ineffective tendency in branch analysis testing is to resort to techniques such as using predicate algebra to map out the required decision values along a path to untested code, or using a debugger to manipulate the decision variables to reach the untested code. These techniques make the assumption that the decision predicates are all correct. This is an invalid assumption. In fact, in many cases the majority of code errors are in the decisions themselves, not the straight-line code between them.

This is borne out in data that was compiled for the components of a networking product developed at HP (Fig. 1). There was a 66% increase in error density from the five components with the lowest branch density to the six components with the highest branch density. This shows that the techniques described above have only limited degrees of usefulness. Since mapping out decisions is actually a way of manually executing the code, some of the decision errors may still be found. In fact, this is a degenerate form of a walkthrough. However, in this form it is a highly error-prone technique. In the case of using a debugger to force the decisions, there is no way to ensure that artificially changing decision variables won't hide or even cause errors. The main problem is that test data obtained by these techniques has a significantly lower chance of exposing errors than traditional functional testing techniques. Since the objective of testing is to expose any errors that may exist, these techniques should be avoided whenever possible.

New HP Methods

This section describes some new techniques that have been used at HP to help alleviate the problems described above. A comprehensive branch analysis methodology based on these techniques is then described.

Setting Requirements for Branch Analysis

Many of the problems we ran into when we first started using branch analysis were caused by setting high coverage requirements on testing that was being conducted on systems software that was fully integrated. It turned out that this was the wrong place for these requirements to be set. This section describes the issues related to branch analysis through the prerelease testing life cycle. In this section, a procedure is a single subroutine, procedure, or function in the sense that a compiler would see it. A component is a cluster of these procedures that together form a significant functionality of the system. A system represents all of the functionalities that would be delivered to the end user.

Procedure Level Testing. There are few methodologies designed for this phase of testing. This is understandable because this type of testing is almost always conducted by the engineer who wrote the code and it is usually conducted shortly after the code is written. As a result, the tester knows the code very well and can use *ad hoc* techniques to ensure that it adheres to its specifications. This is normally done by creating a driver and stubs around the procedure to be tested. The driver orchestrates the testing while the stubs simulate any outside functionality needed for the procedure to perform its assigned function.

It would be a relatively simple task to conduct branch analysis on software of this type. The tester knows the functionality of all of the code and there are very few branches. Also, error conditions can be forced or simulated by using stubs and global variables.

Unfortunately, testing at this level has such a limited scope that any branch analysis data that is obtained would be suspect. The reasons for this are fairly simple. The testing is being conducted in an unrealistic environment in which no interfaces are tested. As a result, none of the invalid assumptions that each engineer has about these interfaces will be exposed by the testing. Also, since we are dealing with a very small piece of code, it would be trivial to attain 100% branch coverage at this level. This leads to a false sense of security that there are no errors in the procedure. Therefore, we do not recommend imposing branch analysis requirements on testing at this level. Please note that we are not recommending skipping procedure level testing, rather we are recommending conducting procedure level testing without *requirements* for branch analysis.

Component Level Testing. Testing at the component level is conducted in much the same way as at the procedure level. The driver and stubs are used in a similar way. The driver will necessarily be more complex because there is more functionality being tested. Also, there will be fewer stubs but some of them will also have to be more complex.

Branch analysis will still be fairly simple at this level. The code should be compact enough to allow the tester to understand, to some degree, all of its functionality. Also, the driver and stubs can still be used to force or simulate most error conditions.

Although testing at the component level is conducted in a similar fashion to that at the procedure level, the value of the tests is significantly increased. This is because many of the interfaces that were simulated at the procedure level are replaced with the actual code. There are still some missing dependencies, but they can be minimized by a careful partitioning of components. In addition, this is the first level at which meaningful functional testing can be performed. This is because some user-level functionality now exists. A tester can now use the documentation of that functionality to create functional tests. These points add a good measure of test validity to any branch analysis data obtained during testing and make this level a prime candidate for imposing high branch analysis requirements.

System Level Testing. Testing at the system level requires a different strategy. The driver and stubs are eliminated, creating a need to write tests that run the system in the same ways that the end user is expected to run it. Normally the system is built up slowly by adding new functionalities gradually and running tests to make sure the new functionalities work as expected with the rest of the system. As the system is built up, so is the test suite. When the system is complete the entire test suite is run again to make sure nothing was missed.

Branch analysis at this level can be slow, tedious, and frustrating. The main reason for this is obvious; there are more levels of program structure that must be executed to reach branches at the lower levels of each component. Naturally, it is much easier to write a test to execute branches two or three levels deep in a component than it is to execute those same branches when they are now six or seven levels deep in the system. Also, since the driver and the stubs have been eliminated, it can be very difficult to test error detection and recovery code. This becomes even more difficult when testing systems software, in which some of the errors are hardware or timing related. One other serious problem at this level is that this testing occurs immediately before the product is released. There is usually a great deal of pressure on the testers to get the product ready to ship. While the branch analysis requirement does not allow the testers to forego quality objectives in deference to schedule pressure, it does put added pressure on these engineers because there is a branch coverage requirement that now must be met.

Test validity is highest at this level of testing. There are two main reasons for this. The first is that the code is all there. There are no longer any artificial interfaces that can hide errors. The second is that, at this level, tests are being run that execute the system the way the end user is expected to execute it. Therefore, the tests can find documentation and usability problems as well as coding errors. In any event, branch coverage at this level should be expected to drop from that achieved at the component level. Emphasis should be placed on testing intercomponent functionalities and interfaces. The intracomponent functionalities have already been thoroughly exercised during component level testing.

These recommendations are designed to provide a guideline for the cost-effective use of branch analysis. It is crucial that the ultimate goal of testing remain the detection of errors. We have seen that imposing high branch analysis requirements on system level testing can alter the priorities of the testers because the product is about to be released and the branch analysis requirements have not yet been met. When this happens, branch analysis testing is no longer cost-effective and other methods of software verification should be employed.

Team Testing

A way to avoid the problem of misinterpretation of the branch coverage metric is to try to use only functional testing techniques to augment the test suite. However, branch analysis tends to give the tester an understanding of very low-level code, and it is difficult to ignore this knowledge when creating functional tests. A technique known as team testing, which is being used for testing the MPE XL operating system, has been very effective at resolving this problem. This technique uses two engineers in a way that separates the low-level knowledge of the system from the design of the functional tests.

The engineer who is familiar with the code under test is referred to as the expert. The other engineer is referred to as the tester. Initially, the tester designs functional tests for the software. After the tests have been run, the branch coverage data is examined by the expert. The expert is then in a position to determine what functionality of the system has not been tested by the initial test suite. The expert then recommends additional functional tests to the tester, along with an estimate of the expected improvement of branch coverage. The tester then takes the functional specification of the test and uses traditional functional testing techniques to create a series of tests that will thoroughly test that functionality. This process is repeated until testing is complete.

The choice of the expert for the team testing technique is very important. It is the expert who is responsible for determining the functionality of individual sections of the code. The expert must therefore be very familiar with both the product being tested and the code being tested. The difference between the product and the code is that the former implies general knowledge about the product's functionality and the latter implies specific knowledge about how the programmers have implemented certain features. The best choice of expert is one of the original authors of the code.

Since the tester will be responsible for designing and implementing functional tests, he or she must be familiar, and preferably experienced, with the functional testing techniques described earlier in this paper. The tester must also be familiar with the functionality of the product being tested and with the operation of the hardware on which the product is implemented. An understanding of the ultimate user of the product is also needed. The quality assurance engineer usually fits into this role quite well.

Proper communication between the expert and the tester is essential for this team testing process to work. The expert must have a way of easily and efficiently communicating the necessary information about the functional testing that must yet be done on the software. If the communication medium is too cumbersome or too inefficient then it simply won't be used and the full benefits of team testing will not be achieved. Fig. 4 shows the information flow for this process. The tester gives the expert a detailed report indicating the branch coverage of the tests so far. The expert examines the data and gives the tester a set of forms that indicate the additional functionalities that need to be tested and the expected increase in branch coverage. The latter

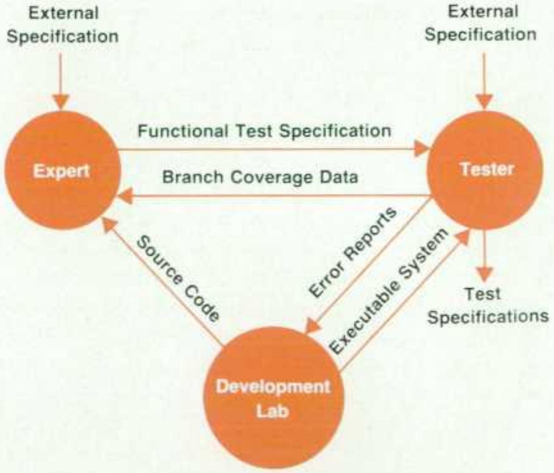


Fig. 4. *Information flow in team testing.*

piece of information is used by both engineers as a checkpoint for proper communication.

The expert has the branch coverage information obtained from the tester and the source code for the software being tested. Using this information, the expert can determine which physical portions of the code have not been tested by the test suite so far. By examining the untested portions of the code, and understanding the functional specification of the software, the expert determines what new functional tests need to be added to the test suite. While doing this, the expert should write down a list of previously unexecuted branches that these tests will execute. The expert is not concerned with the details of writing or executing the tests or any other details of testing. The expert then passes the branch "hit list" along with the functional test descriptions to the tester.

The tester receives the functional test descriptions from the expert and proceeds to write functional tests that fully test the specified functionality. This is the critical part. If the tester does not test the functionality using all of the standard functional testing techniques, then all of the effort going into the testing will be of limited value. This is because these functional testing techniques are designed to find errors, whereas branch analysis is designed to ensure that all of the code is executed. If the tests can be fully automated, the tester adds them to the automated test suite. If the tests cannot be fully automated, a careful log is kept indicating the steps necessary for the test to be replicated. In any event, the tester then creates the necessary environment for the functional test to take place and executes the tests. If any of the branches on the hit list are not executed, the tester must notify the expert of this discrepancy. This could be either an error or simply a miscommunication between the expert and the tester. Either case must be resolved. The new branch coverage data is then given back to the expert and the process is repeated.

As already mentioned, there are several areas that need special attention for team testing to be successful. First, the members of the team must be chosen carefully. A weak link in this chain could destroy any benefits the methodology has to offer. Next, the lines of communication between the tester and the expert must be clear and efficient. Both engineers can be excellent, but if they don't communicate properly, neither will be effective. Finally, the tester must test the software using functional testing techniques, independent of branch analysis. It's not enough just to execute a branch. The branch must be tested in a way that will expose errors.

Branch Analysis Walkthroughs

Now we introduce a technique known as a *branch analysis walkthrough*. The goals of the branch analysis walkthrough are vastly different from those of traditional walkthroughs and inspections. In fact, branch analysis walkthroughs may be more aptly termed risk analysis walkthroughs. Branch analysis walkthroughs are used to identify critical, complex, and error-prone code that should be targeted for later verification. This makes it possible to direct the remaining resources of the project toward the verification of critical code or code that has a reasonable possibility of containing an error. This provides a much more cost-effective software verification process.

Walkthroughs and code inspections offer many benefits (see box, page 21). Since there are plenty of references on traditional structured walkthroughs and inspections[1,3,4,5,8] we will only discuss the key differences between a branch analysis walkthrough and the more traditional approach discussed in the literature.

Two prominent features of the branch analysis walkthrough are that the code being inspected is sparsely dispersed throughout a software component, and that there is usually a lot more code needing inspection. Fortunately, since the goal is simply to perform a risk analysis on the code, the walkthrough team is capable of inspecting the code at a high rate. However, this is not true of the preparation stage. For the branch analysis walkthrough to be successful, it is critical that all members of the team be well prepared.

Preparation. There are four steps in the proper preparation for a branch analysis walkthrough:

1. Test the code as much as possible. Since you will have to walk through all of the untested code, it is very important that the volume of code be minimized.
2. Have the moderator annotate a listing. The moderator should take a current listing of the code and mark all branches that have not been executed during testing. While doing this, any interfaces into or out of this code should be cross-referenced unless they are already easy to find.
3. Hold a preview meeting. A preview meeting should be held at least one week before the walkthroughs are to commence. The team should handle the following items during this meeting:
   - Pass out copies of the annotated listing to all of the team members
   - Organize the walkthrough process
   - Determine the logical order in which the branches will be inspected
   - Develop a schedule for completing the walkthroughs
   - Clarify each engineer's role in the walkthroughs.
4. Analyze the code. Each member of the team must become very familiar with the code to be inspected before the walkthroughs begin. This is critical because during the walkthroughs themselves the team will be moving fairly quickly.

The Walkthroughs. To make the branch analysis walkthrough move quickly and smoothly, each branch is simply categorized, and only a small portion of the branches are actually verified during the walkthrough. The categories that branches can be put into are:

1. Testable. A test case can be created to execute this branch without an unreasonable amount of effort.
2. Signed off. The branch is relatively trivial, correct, and not critical to the overall function of the system.
3. Not signed off. An error was found in the decision logic or the code for the branch.
4. Unreachable. Previous logic has eliminated the possibility of executing this branch.
5. Verification necessary. The branch is too critical or too complex to be signed off without more thorough verification.

Testable branches are typically functional tests that were overlooked. These branches can usually be identified very easily and classified quickly. For a branch to be signed off, the walkthrough team must feel very confident that the code is correct. The branches in this category are typically simple error-exit code that cannot be tested easily because the error must occur during a very small time interval. These are the only branches that are completely verified by the branch analysis walkthrough. Virtually all unreachable branches fall into one of two categories. They are either redundant error checking or hooks inserted into the code to facilitate the addition of scheduled enhancements. In the case of a redundant error check, if the error being tested cannot possibly occur between the two error checks then the redundant one should be removed. The rest of the branches should be left in the code but deducted from the calculation of the coverage. The last category is used to speed up the branch analysis walkthroughs. Any time a branch would require considerable effort for the team to be confident that the branch is correct, or if an error in this branch could seriously degrade the reliability of the entire system, this branch needs to be investigated more thoroughly at a later date.

As you can see, these categories are designed to allow the walkthrough team to move quickly. Most time is spent verifying branches that are relatively trivial (those in categories 2 and 3). The rest of the categories are to be dealt with later, and therefore each branch can be scanned quickly to determine the proper category.

It is important that the members of the team have a high level of concentration during walkthroughs. This is even more important for branch analysis walkthroughs because of the speed at which the code is being inspected. To achieve this, the walkthroughs should be held in a quiet place with no outside distractions and limited to one two-hour session per day. Research shows that continuing beyond two hours can seriously degrade the quality and efficiency of the walkthroughs.[8] It is the responsibility of the moderator to ensure that these guidelines are adhered to.

Follow-Up. During the branch analysis walkthrough, a follow-up form must be opened for each branch not put into the signed-off category. All of these branches must eventually be signed off before the product is released. The follow-up form contains the following information:

- The branch number and location
- The category the branch is put into
- Comments on why the branch was put into that category
- Follow-up recommendations
- Follow-up comments
- Signature and date of sign-off.

The first three of these are filled out by the moderator during the branch analysis walkthrough. The fourth is filled out by the team before follow-up commences. The last two are filled out by the quality assurance engineer assigned to sign off these branches. It is important that the branch number and location be precise enough so that if the code changes between the time of the walkthrough and the follow-up, the branch in question can still be identified. The comments on the category decision should at least include: a description of the error for not-signed-off branches, a functional test description for testable branches, and any other information that will help the follow-up engineer to verify whether or not the branch is correct.

Follow-up recommendations are essentially a suggestion by the team on how the branch should be verified. This is explained further in the next section. The follow-up should include:

- Verification of fixes for errors found. Special care should be taken on follow-up of errors found. Studies show that one out of every six code fixes is either incorrect or introduces other errors.[8]
- Verification that tests were run to execute testable branches.
- There are two possible resolutions for each of the unreachable branches. If it is a hook, remove it from the calculation of branch coverage. If it is unreachable, remove it from the code.
- Obviously, branches in the verification-necessary category must be verified by whatever formal means are deemed appropriate and then signed off.

Resources Needed. Table I shows the recommended

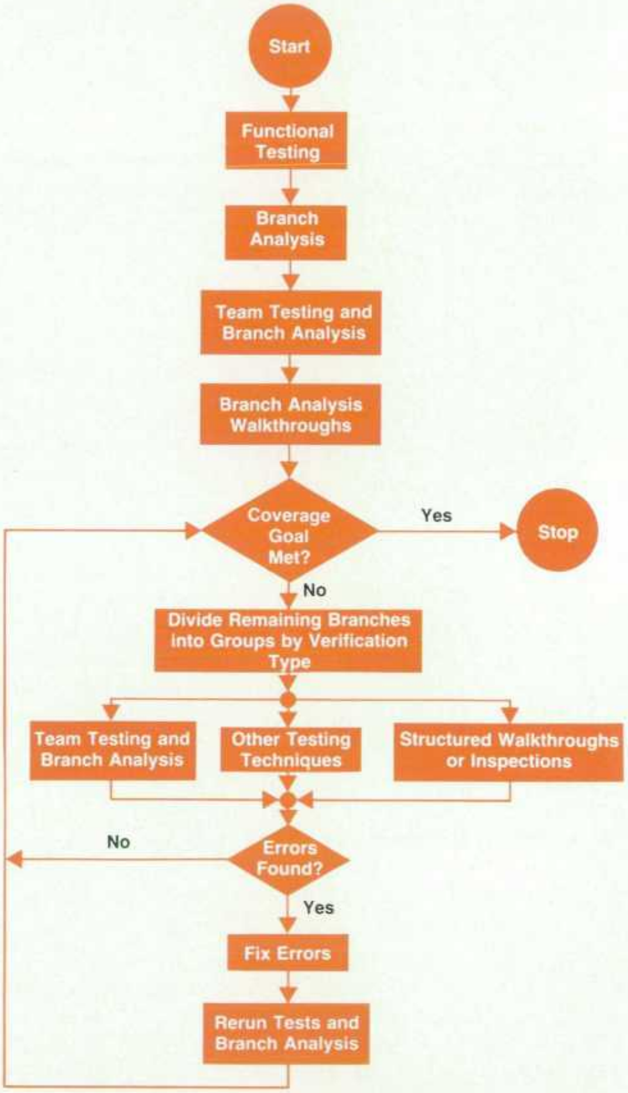

Start
Functional Testing
Branch Analysis
Team Testing and Branch Analysis
Branch Analysis Walkthroughs
Coverage Goal Met?
Yes
Stop
No
Divide Remaining Branches into Groups by Verification Type
Team Testing and Branch Analysis
Other Testing Techniques
Structured Walkthroughs or Inspections
No
Errors Found?
Yes
Fix Errors
Rerun Tests and Branch Analysis

Fig. 5. *Flow chart for a comprehensive verification methodology based on branch analysis.*

Advantages of Code Inspections

In the past, there was a great debate on the use of code inspections as a software verification tool. It was fairly obvious that the technique could be effective but there were concerns. They included:

- Cost. Because code inspections are highly labor-intensive, they are exceptionally costly.
- Benefit is long-term. It is difficult to determine the effectiveness of the inspections until long after the product is released. Human nature is to avoid a short-term cost if there is only a long-term benefit.
- Reproducibility. There is no way to automate a code inspection and therefore it is extremely difficult and costly to try to reproduce the code inspection for regression verification.
- Consistency. It is very difficult to ensure consistent error detection efficiency over time and especially from person to person.

More recently the debate slowed. Most people understood and believed in the effectiveness of code inspections but still had reservations because of the cost-effectiveness issue. The common excuse was "I believe that code inspections are good, but they take so much of my engineers' time and are so expensive that I can only afford to use them sparingly." This was common because there was little or no statistical data that showed just how cost-effective code inspections really were. This is no longer the case. References 1 and 3 show clearly that there is now plenty of experience and statistical data verifying that code inspections are highly cost-effective.

One reason that code inspections have been proven cost-effective is that they have many beneficial side effects other than increased quality.[1-5] These include:

- Improved readability of code. Since the code must be read by several engineers other than the implementor, the code must be very readable before the inspection can commence.
- Training. Because several engineers are conducting a critical evaluation of the code, there is usually a positive exchange of technical information and interesting programming techniques and algorithms during the inspection.
- Insurance. Understanding of the code is disseminated during the inspection. This is valuable if the developer leaves the project before it is finished. Someone else on the inspection team can pick up where the developer left off relatively quickly.
- Morale. Morale can be improved in two ways. First, inspections break up the engineers' routine with an opportunity to interact with and learn from the rest of the team. Second, no one likes to find errors during testing. The sooner errors are found the easier they are to fix.
- Errors are found, rather than symptoms. When a test fails, all an engineer has to go on is the symptoms of the failure. The engineer must then debug the system to find the error. During an inspection, the error itself is found, making the repair that much easier.
- Many errors are found at once. During an inspection, many errors may be found in the code before the inspection is over. All of these errors can then be fixed at once afterward. It is very rare for a test to expose multiple errors.
- Errors can be prevented. Inspections give the engineers involved an awareness of how errors are introduced into the code. This can help reduce the number of errors these engineers introduce into subsequent coding efforts.

Most of these benefits can be realized immediately, making this technique very cost-effective even in the short term.

References

1. M. Fagan, "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, Vol SE-12, no. 7, July 1986.
2. R. Fairley, *Software Engineering Concepts*, McGraw-Hill, New York, 1985.
3. G. Myers, *The Art of Software Testing*, Wiley-Interscience, New York, 1979.
4. R. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 1982.
5. E. Yourdon, *Structured Walkthroughs*, Prentice-Hall, Englewood Cliffs, N.J., 1979.

Dan Herington
Software Engineer
Information Networks Division

amount of time to perform each of the steps in the branch analysis walkthrough process. Recommendations found in the literature are typically designed for full-scale walkthroughs and are expressed in lines of code per hour. Branches per hour is a better measure because it takes into consideration the branch density of the code. Code that has high branch density tends to be more difficult to understand and more error-prone (see Fig. 1). Therefore, this code should be inspected at a slower rate. Based on our branch density data, the numbers in Table I are roughly equivalent to those in the literature for the preview and preparation rates.

Table I

Preview Meeting	½ hour + 100-150 branches/hour
Preparation	15-20 branches/hour
Walkthrough	30-40 branches/hour

The branch analysis walkthroughs themselves are designed to move quickly and therefore can move at roughly twice the rate of a structured walkthrough or inspection. We have found that the majority of the branches that make it to the branch analysis walkthroughs are very short and simple, making these numbers relatively conservative.

A Comprehensive Verification Methodology

Fig. 5 shows a revised structural testing process that represents how all of the techniques described above fit together to form a coordinated software verification process. This process will maximize the cost/benefit ratio for conducting structural software verification.

Team testing should be conducted before the branch analysis walkthroughs for two important reasons. The first reason is that walkthroughs are a labor-intensive activity. This means that they will be more expensive and more error-prone than testing. Therefore, it is important to make sure that the volume of code that must be inspected in the walkthroughs is minimized. The second reason is that team testing creates an automatable verification of the functionality of the software. This becomes very important when the time comes to conduct regression verification. Since the walkthroughs can't be automated, it is not likely that they will be redone during regression verification.

After the branch analysis walkthroughs, any branches not put into the signed-off category must be verified by some formalized means. The decision on whether to use a team testing approach or to conduct traditional structured code walkthroughs should be made on a per-branch basis. The team that conducted the branch analysis walkthroughs is probably the best judge of which of these techniques is most appropriate for each branch.

Once it is decided how to verify the remaining branches, the teams can then go on to the formal verification. Since these are the branches that were deemed important enough to warrant formal verification, the verification should be thorough and as reproducible as possible. If any errors are found during this verification, they should be fixed and the regression package augmented and rerun using branch analysis. The release criteria are then checked and the loop is repeated until they are met.

Acknowledgments

The authors would like to thank Todd Whitmer for his insights in the creation and development of the branch analysis walkthrough methodology. We would also like to mention Dan Coats for his work in the development of the team testing approach for branch analysis.

References

1. R. Fairley, *Software Engineering Concepts*, McGraw-Hill, New York, 1985.
2. W. Howden, "Functional Program Testing," *IEEE Transactions on Software Engineering*, Vol. SE-6, no. 2, March 1980.
3. G. Myers, *The Art of Software Testing*, Wiley-Interscience, New York, 1979.
4. R. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 1982.
5. D. Richardson and L. Clarke, "Partition Analysis: A Method Combining Testing and Verification," *IEEE Transactions on Software Engineering*, Vol. SE-11, no.12, December 1985.
6. E. Weyuker and T. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," *IEEE Transactions on Software Engineering*, Vol. SE-6, no. 3, May 1980.
7. T. DeMarco, *Controlling Software Projects*, Yourdon Press, New York, 1982.
8. M. Fagan, "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, Vol. SE-12, no. 7, July 1986.
9. E. Yourdon, *Structured Walkthroughs*, Prentice-Hall, Englewood Cliffs, N. J., 1979.

Reader Forum

The HP Journal encourages technical discussion of the topics presented in recent articles and will publish letters expected to be of interest to our readers.
Letters must be brief and are subject to editing.
Letters should be addressed to:

Editor, Hewlett-Packard Journal, 3200 Hillview Avenue,
Palo Alto, CA 94304, U.S.A.

Editor:

The viewpoint expressed by Zvonko Fazarinc in the March issue of the *HP Journal* is quite unorthodox and, in that respect, quite stimulating ("A Viewpoint on Calculus," p. 38). Of course Professor Fazarinc is right when he claims that "we live in a computer era" and "we can solve only a handful of differential equations."

But is it really what we want? On more than one occasion what we need is a qualitative discussion of the system dynamics: is the asymptotic solution periodic, divergent, or punctual? Moreover, it is very often useful to discuss how the parameters affect the solution. For instance, in fluid dynamics the coupling of gravity and diffusion leads to nontrivial effects. These effects would be completely blurred by a brute force calculation, even though the student would gain skills by developing the algorithm.

As a conclusion, while I totally agree that a numerical solution is certainly a solution, I claim that the development of numerical techniques should reinforce rather than weaken the teaching of system dynamics (or differential geometry).

Alain Maruari
Professor
École Nationale Supérieure des
Télécommunications

I have no doubt that study of qualitative system dynamics provides a good stepping stone toward an understanding of associated phenomena. At the same time I must state that the evolutionary discrete mathematical formulation coupled with a good graphics interface in an interactive environment provides by far the best medium for such studies.

I have spent two years among students and faculty at Stanford University exploring the potential of computers for building intuitive understanding of new concepts. This research has strengthened my belief that discrete mathematics offers considerably more than just "brute force calculations." If it is used to express first principles it has the potential to mimic nature. If it is given an appropriate visual presentation and associations with some familiar phenomena, it can shorten the time normally needed for acquisition of intuitive understanding of new concepts. Given a chance to evolve in time, it can provide us with an insight into quantitative or qualitative dynamics that has no equal anywhere.

It is difficult to illustrate these points in a general way, so allow me to use your example of gravity and diffusion as a vehicle. I have developed a teaching module at Stanford that addresses these phenomena in a purely qualitative way but uses discrete mathematics behind the scenes. Instead of presenting the student with the equation

$$\frac{\partial C(x,t)}{\partial t} = \frac{\partial}{\partial x} D(x) \left[\frac{\partial C(x,t)}{\partial x} - \frac{F(x,t)}{kT} C(x,t) \right]$$

which separates the average mortal from those who have mastered the partial differential calculus, the module first makes an association with the mechanical world. It does so by presenting a number of colliding particles moving away from the "crowd," where the collisions are more frequent but result in a less impaired motion in the direction away from the high density. The number of particles in a given position is plotted dynamically in the form of a graph directly above them. This provides an almost instantaneous intuitive understanding of the process and helps visualization later when more complex problems are addressed without showing the particles.

Dynamic displays of concentration C(x,t) as a function of space and time evolving under the influence of gradients and external forces F(x,t) may be observed. The student can enter arbitrary diffusivity D(x), force profile F(x), and initial concentration C(x,0) via graphic inputs. Simultaneous display of fluxes provides another aid for visualizing nontrivial cases.

Students have been generating the earth's atmosphere by entering inverse-square force profiles. They have produced nonspreading concentration packets by shaping diffusivity profiles appropriately. But they have definitely gained a strong intuitive understanding of asymptotic behavior without missing the transient behavior, which contains powerful clues to understanding the phenomena.

A number of other modules have been developed that cover fields such as semiconductor physics, short-range and long-range forces, transmission lines, wave interference phenomena, and random processes. All of them have proven to provide insight into qualitative behavior without the simplifications required when infinitesimal formulations are used.

*My paper, "Computers in Support of Conceptual Learning," presents a short description of motivations, results, and conclusions derived from the Stanford study.**

Zvonko Fazarinc
Senior Scientific Advisor for Europe

*Available from Zvonko Fazarinc, Hewlett-Packard Company
1501 Page Mill Road, 3U, Palo Alto, California 94304, U.S.A.

Authors

June 1987

4 Graphics Tablet

Thomas Malzbender

Tom Malzbender joined HP in 1982 upon getting his BSEE degree from Cornell University. His HP contributions include the design of an optical mouse and several related ICs. He also contributed to the analog design and firmware for the HP 45911A Graphics Tablet. His work in these areas has led to two patent applications. He's currently working on neural networks at HP Laboratories. His professional interests include cybernetics, neural computation, and fractal geometry and he has written two papers on the latter topic. Before coming to HP he worked for National Semiconductor Corporation and for Teradyne. Born in Munich, Federal Republic of Germany, Tom now lives in Sunnyvale, California. He enjoys rock climbing and electronic music synthesis.

8 HP-HIL

Robert R. Starr

Rob Starr started at HP in 1979 as a production engineer for HP 262X Terminals and later moved to an R&D position, where he contributed to the development of the HP 2627A Color Graphics Terminal, the HP Mouse, and HP-HIL accessory cards for HP 150 Computers. He's currently working on 3.5 inch flexible disc drive qualification. Rob is a California native with a 1979 BSEE degree from California Polytechnic State University at San Luis Obispo. He was born in Oakland and now lives in San Jose. His outside interests include photography, skiing, aviation, woodworking, and real estate. He says he'd like to build his own home someday.

13 Branch Analysis

Paul A. Nichols

With HP since 1972, Paul Nichols is productivity manager for the Information Networks Division. Earlier he was software quality manager for the HP 300 Computer. A native of Santa Cruz, California, he earned an AA degree in business administration from Cabrillo College in 1969. Before coming to HP he worked for eight years as a systems analyst in the electronic banking field. Paul still lives in Santa Cruz, is married, and has two sons. His leisure interests include photography and sailing.

Daniel E. Herington

Born in Brooklyn, New York, Dan Herington completed work for his BS degree in computer science from Arizona State University in 1985. After joining HP's Information Networks Division the same year, he worked on the development and application of new software testing techniques and tools, including branch analysis. He's a member of IEEE and ACM. Dan lives in Santa Clara, California and enjoys basketball and golf.

Roger D. Lipp

Born in Sarasota, Florida, Roger Lipp is a graduate of Purdue University (BS computer science 1985). He has been with HP since 1985 and is a software tools and productivity engineer for the Information Networks Division. His professional interests include computer-aided software engineering, artificial intelligence, and software reuse. Roger and his wife live in Sunnyvale, California and are expecting their first child in September. He's active in his church and enjoys photography, music, and astronomy.

24 Viewpoints

Yoshio Nishi

With HP Laboratories since 1986, Yoshio Nishi is a specialist in semiconductor device and process technology. He directs the silicon VLSI research laboratory for the Circuit Technology Group and previously headed the Semiconductor Device Engineering Laboratory for Toshiba Semiconductor Group in Japan. He was also a visiting professor at Hokkaido University and is a consulting professor at Stanford University. He is named inventor for about 50 patents related to semiconductor processes and device structures in the United States, Japan, and Europe. In addition, he has authored approximately 50 papers and is contributing to four books. Born in Yokohama, Japan, Yoshio received his BS degree in metallurgy in 1962 from Waseda University and his PhD in electronic engineering in 1973 from the University of Tokyo. He and his wife and two children currently live in Palo Alto, California. His leisure-time activities include playing violin and piano, listening to music, skiing, and visiting museums.

26 UNIX IPC

Marvin L. Watkins

A California native, Marv Watkins holds a BA degree in psychology and an MS degree in mathematics from San Jose State University (1975 and 1978). He also studied computer science at Ohio State University and received his MS in 1979. With HP since 1986, his major contribution has been a UNIX-system-based workcell controller for a surface-mount manufacturing line. His professional experience before coming to HP was varied. Among other accomplishments, he developed data base managers and a graphics spooler and verified real-time software and firmware for an electronic flight instruments system. He also served in the U.S. Army. He's author or coauthor of two articles on software testing and computer displays. Born in Santa Barbara, Marv is now a resident of Los Gatos. In addition to his interest in software engineering, he enjoys jazz, science fiction, house plants, and exploring the backcountry.

Direction of VLSI CMOS Technology

by Yoshio Nishi

SINCE STOCHASTIC FLUCTUATION of device and process parameters becomes more significant with increasing numbers of transistors on a chip, there is a strong requirement for increased noise immunity and decreased power consumption in higher density circuits. Although low-power CMOS circuits were invented in the 1960s, they did not increase in importance until integration density exceeded 100,000 devices/chip. Since then, CMOS has penetrated into static memories as well as microprocessors. This article will briefly review the current status of CMOS technology and discuss engineering challenges for future microcircuit technology.

CMOS Integrated Circuits

The earlier CMOS integrated circuits started with calculator chips, watch and clock circuit chips, etc., which required lower power consumption because of the limited capacity of the battery cells available for these applications. CMOS circuits are also widely used for logic gates, analog-to-digital converters, and phase-locked loop synthesizers, which require either large driving capability, wide operating voltage, or noise immunity. Now the application of CMOS has been expanded to cover memories and microprocessors for computing engines.

The first CMOS VLSI circuit ever made was a 16K-bit static memory which certainly enjoyed all of the features of CMOS mentioned above. Since then, each generation of memory device ICs has increasingly used a higher percentage of CMOS devices. About half of the 64K-bit static memory chips were made using CMOS technology and all 256K-bit static memory has been built in CMOS. Even in the case of dynamic memory, which has been implemented using NMOS technology because of its strong requirement for the highest density, CMOS is used for more than 40% of the current 1M-bit memory chips. The 4M-bit and 16M-bit memory chips described at the 1987 International Solid-State Circuits Conference have verified again the strong thrust toward CMOS technology. A similar change has occurred in the logic VLSI area in that most of the newer chips have been implemented as CMOS circuits for the same reasons described above.

Recent progress in application-specific ICs has been made based upon CMOS technology. These ICs are considered as one of the most important device technology application areas for a company that has captive VLSI capability for internal use only.

Active Devices

A CMOS circuit cell consists of a pair of MOSFETs, one n-channel and one p-channel. Traditionally, n-channel MOSFETs have been made starting in a p well in an n substrate. Later, the use of p-channel MOSFETs, formed in n wells in a p substrate, became common, especially for EPROMs, microprocessors, and dynamic RAMs. Since the scaling down of device geometry for higher performance requires high substrate density even with structured-in-depth impurity profiles, the twin-well structure, a p well for NMOS and an n well for PMOS, was introduced because of its larger degree of freedom for channel impurity doping and lower junction capacitance design. An n-channel MOSFET needs special consideration to avoid hot-electron-induced degradation when channel length becomes smaller, so that the increased electric field strength near the drain does not cause impact ionization of electron-hole pairs. The lightly doped drain (LDD) FET structure has been introduced to lower the drain electric field.

On the other hand, fabricating higher-density p-channel MOSFETs becomes more difficult because acceptor impurity diffusion in silicon is much faster than donor impurity diffusion, which makes diffusion of shallow p junctions more difficult. The use of boron-fluoride (BF) implantation and rapid thermal annealing process steps has become quite common for this reason.

Oxide integrity is another major concern for both NMOS and PMOS devices. A recent 4M-bit dynamic memory uses 10-nm-thick oxide for the memory cell capacitor and 15-nm-thick oxide for the active gate insulator. To increase performance and circuit density, these values should decrease further, which will cause interesting physical problems and technical difficulties. Threshold voltage control would be one of the most difficult issues because widely used channel doping techniques might not provide acceptable device performance. The increase in source resistance with decreasing junction depth almost forces us to apply some kind of more-conductive layer on source and drain regions. The most commonly investigated material for this purpose is titanium silicide. Other materials such as platinum silicide, cobalt silicide, and titanium nitride have also been investigated.

The downward scaling of device feature sizes to less than a 0.5-micrometer channel length has led to the serious consideration of operating CMOS devices at lower temperatures. The associated benefits of lower subthreshold leakage currents, steeper logarithmic current-voltage characteristics, and higher mobility promise higher-speed performance. Lower temperature operation eventually offers a chance to obtain radically improved total VLSI system performance by combining it with "high-temperature" superconductive interconnections as discussed later.

Interconnections

As the speed performance of active devices has been improved, the influence of interconnection technology on chip performance has become significant. Aluminum doped with silicon and/or copper is the most popular material for both signal and power supply lines. The key parameters in this area are density, signal propagation delay, and current-carrying capability. Density refers to the number of interconnection layers and the metal line widths and spacing. Propagation delay is mostly determined by RC constants. Current-carrying capability is a strong function of electromigration endurance along the interconnection lines and at contact/via regions.

The basic feature of CMOS, low dc power consumption, generally represents the superiority of CMOS over other technologies, but for high-frequency operation most of the power dissipated on a chip is ac power, which even CMOS cannot do anything to reduce. Once we take a careful look at a VLSI chip, 70% of the ac power dissipation occurs at the circuits that drive larger capacitive loads and switch simultaneously, which most likely happens at output driver and buffer circuits. Temperature increases because of higher current density in interconnections and this results in increased electromigration effects. Thus, the choice of interconnection material must be carefully made to meet several conflicting requirements. Several circuit tricks, which can effectively decrease the power dissipation at driver circuits, have substantial importance, and bipolar-CMOS circuits could be one of the possible solutions.

When one tries to think of an ideal interconnection, its properties have to include zero resistance with infinite current-carrying capability. Recent significant progress in developing materials that are superconductive at relatively high temperatures (77K) might open a new horizon in this area. Superconductive interconnections may add another value, such as automatically solving some inductance problems because of superconductivity's natural diamagnetism.

Although there will be many important and essential improvements in VLSI circuits using this new technology, we have to look carefully at the potential issues that may arise. Among them are how closely the thermal expansion coefficients of the new materials match that of silicon, what will be the critical current density where superconductivity collapses, how to control compositional uniformity of the new materials, etc.

Isolation

To isolate one active device adequately from other devices in monolithic integrated circuits has been the largest concern for higher-density integration. MOS designs took advantage of not needing any substrate isolations like bipolar integrated circuits. CMOS, however, has two different active devices in a circuit cell so that there must be substrate isolation. Even in NMOS integrated circuits, how to minimize the distance between one FET and adjacent FETs has been and continues to be a major engineering challenge. The local oxidation of silicon (LOCOS) in selected regions as an isolation process had been the solution for a long time. However, the bird's beak (an undesired intrusion of the oxidized silicon into adjacent device regions) characteristic of the LOCOS process is not allowable any longer for VLSI dimensions of a micrometer or less. There have been a large number of approaches to minimize the interdevice distance. Most of them try to etch a trench or moat around the active devices and then backfill it with insulating materials.

Another approach is to use an insulating substrate instead of silicon, a technology known as silicon on insulator (SOI). One example in the past that falls into this category was the silicon-on-sapphire (SOS) structure. Several experiments to realize a multilayer silicon/insulator structure have been tested for three-dimensional integration, and such studies are one of the most interesting research areas for the ULSI era.

Technology Driver

Technology progress in integrated circuits has not been done with a smooth continuous trajectory. Rather, it has been more like climbing up a stairway. Every two to three years, a new technology generation has been introduced, and this most likely will continue until silicon VLSI/ULSI hits the ultimate barrier at which one will have to switch to other active device materials for further progress.

Each technology advance has required that a variety of technologies be integrated into one set of processes by which we can build desired circuits. There must be some vehicle that can continuously stimulate and finance advances toward higher density, higher performance, and lower cost. This vehicle, called a technology driver, has been dynamic memory for a long time. Sometimes static memory has served this purpose, predominantly for CMOS technology evolution.

The question may arise, "What will be the technology driver in the future?" There is a strong potential that future microprocessors and their associated families of chips will include more memory capacity on each chip, and that memory chips will include more logic functions. Thus, since dynamic memory tends to have quite a unique structure for a cell, which is clearly apart from other chip technology, it is somewhat unlikely that dynamic memory development will continue to drive IC technology evolution. The fact remains that both dynamic memory and static memory will require higher FET density per chip in the future and that VLSI logic circuits will require higher interconnection density and an increased number of metal layers.

A significant aspect of the previous technology driver, dynamic memory, is that each generation's production and sales have provided the R&D funds essential for the evolution of IC technology itself. The R&D period for one technology generation used to be three years, but is now increasing. Yet, every new technology generation still appears every two to three years. This is made possible by overlapping the R&D schedules for successive technology generations and puts an increased emphasis on selecting technology drivers that will provide the funds to support further R&D.

Hence, there will not be any unique choice for a technology driver under diverging system requirements. However, it seems natural that a more system-oriented company will choose static memory as a technology driver with the expectation that such technology will migrate in a more flexible way into microprocessors and application-specific ICs, which can make highly value-added products through which essential R&D funds will be generated.

At this time, CMOS is apparently the most rapidly progressing technology with static memory and microprocessors becoming its technology drivers. Although there is great potential for CMOS to meet most system requirements—speed, performance, integration density, power consumption, and noise immunity—one can easily foresee continuing engineering challenges as well as device physics problems.

An exciting decade for semiconductor technology is ahead, particularly for CMOS VLSI/ULSI. From the viewpoint of the Hewlett-Packard Company, how HP drives its own technology to be in a worldwide leading-edge position will certainly affect most of the company's products. The key is how HP can successfully leverage system performance by integrating state-of-the-art VLSI/ULSI technology. The importance of putting the right combination of unit technologies together to maximize performance and optimize system positioning will be increasing as silicon devices get closer and closer to their ultimate performance barrier.

Software Architecture and the UNIX Operating System: An Introduction to Interprocess Communication

Signals, pipes, shared memory, and message queues are some of the facilities provided by the UNIX® operating system for communication among software modules. The strengths and weaknesses of each facility are discussed.

by Marvin L. Watkins

SOFTWARE COMMUNICATIONS, the exchange or sharing of information by computer programs, is related to several central software architectural concepts. Fig. 1 shows these relationships and provides a road map of the ideas this paper will touch upon. The discussion is organized into three major sections. The first section deals with some fundamental ideas and concepts of software engineering that arise in multiprocessing systems. Here, the important concepts of complexity, modularity, concurrency, and synchronization are presented. The second section deals with use and performance issues that arise with the UNIX operating system's interprocess communication (IPC) facilities. In this section, the UNIX IPC facilities are ranked for various uses and data is presented to support the ranking. The third section discusses each IPC facility in detail.

Fundamentals

We like to think of an activity that a computer is intended to model as a problem the computer's program(s) must solve. The problems that computers are called on to help solve range from the very simple, for example keeping a list of telephone numbers, to the very complex such as fully automating and integrating the diverse machines of a large manufacturing facility.

In general, simple activities generate simple problem statements. These, in turn, generate simple, self-contained programs. Conversely, the more complex the activity a program is intended to model, the more complex will be the program or programs required. Of course, complexity increases costs and jeopardizes success.

Complexity

There are two notions of complexity used throughout this paper. The first is that of problem complexity. Problem statements determine the minimum complexity possible for a software solution. For example, automating a manufacturing facility requires a more complex software solution than a computerized phone list. The second notion is that of program complexity. Some programs are more complex than others, even though they may solve exactly the same problem. For example, "spaghetti code" is almost always more complex than an equivalent structured program.

In an ideal world, problem complexity would be the sole linear determinant of cost for computer programs. A problem statement would be generated, software designed and implemented, and the program(s) delivered. In fact, program complexity affects cost in a nonlinear manner. Many, if not most, software development projects consume more resources (i.e., cost more) than the problem statement demands. The culprit is often unnecessarily complex programs caused by poor, inappropriate, or rushed design.

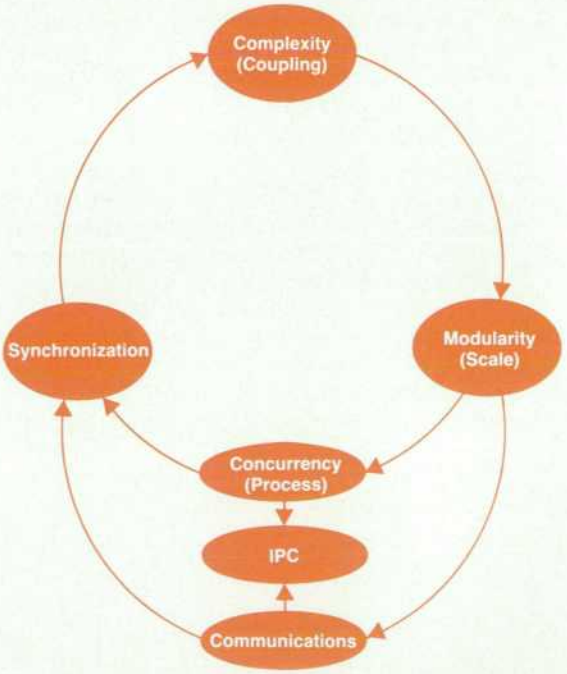


Fig. 1. *Some relations among software concepts. Arrows indicate direction of influence.*

UNIX is a trademark of AT&T Bell Laboratories.

Coupling

Coupling is probably the single greatest contributor to program complexity. In general, the greater the degree of coupling, the greater the complexity of the design and the worse its structure. A change in a module that is coupled to others can have unavoidable effects on them. Even a minor change in a coupled module can have nasty consequences: it can cascade through coupled modules forcing significant changes in many unpredictable ways.

Coupling occurs primarily when the same data or control characteristics or assumptions are embedded in the code of two or more modules. For example, suppose module A performs a series of calculations c1, c2, c3, ... Now suppose module B needs a calculation similar to, say, c2. If A is modified so that B can call A with a new parameter that causes A to execute c2, then A and B have been coupled. Another form of coupling occurs when code has assumptions about external objects embedded within it. For example, a module might assume that a terminal connected to it always displays 24 lines by 80 columns.

When considering trade-offs in program complexity, it is important to consider that tight coupling introduces three major problems: difficulty in finding and correcting design flaws, difficulty in adding new features to a system, and difficulty in migrating functionality to enhance system performance. Loose coupling simplifies maintenance, addition of new features, and upgrades to more computing power.

Modularity

Modularity is among the most powerful methods for limiting a program's complexity to manageable bounds. Modules can exhibit at least three major kinds of relationships or structure: arbitrary, hierarchical, and independent. Arbitrary designs are usually easier and quicker to develop than structured designs. The latter are usually easier to understand, debug, enhance, and maintain. However, additional work must go into the design process to create simple, well-structured, understandable software. That is, near the end of the design phase an unnecessarily complex software architecture is usually produced, which, nonetheless, satisfies the problem statement. More time and resources must be allocated to simplify and organize this design if understandability, flexibility, adaptability, maintainability, etc. are also required. Schedule pressures and, perhaps, inexperience frequently prevent development teams from doing this additional complexity-reducing design work.

Arbitrary designs tend to have enormous amounts of coupling among their modules. Hierarchies and/or independent program structures tend to have much less coupling. However, even they are rarely pure. Even good, structured designs usually have some degree of coupling among their modules.

Top-down design methodologies typically create hierarchical organizations. Such structures model a problem as a hierarchy of functions. Program modules are then created for each of these functions. Hierarchical organizations tend to incorporate program control directly into the problem model (i.e., functional decomposition). This is one of the principal reasons that programs exhibiting such organizations are easier to understand than arbitrary designs.

Some problems do not decompose into hierarchies well. These problems are often characterized by having multiple, asynchronous, concurrent events or processing (e.g., input/output or user query formation and data base search). Such problems can often be modeled as independent functions. These functions can then be implemented as independent programs, provided there are useful communications mechanisms to mediate data exchange.

Modularity is both a recursive and a scalable notion, recursive because modules can contain modules of the same kind and scalable because lower-level modules are usually contained in higher-level ones. For example, instructions can contain instructions, blocks can contain blocks, and procedures can contain procedures, etc. while procedures contain blocks that contain instructions. Fig. 2 compares a scale of software modules with increasingly complex problems that might be solved by software composed of modules at and below each level. The module scale's levels are:

- Instructions. For example, while (*p) p++ and IF (A.GT.B) 10,20,30.
- Blocks. For example, begin ... end and {...}.
- Procedures. For example, C procedures and Fortran subroutines.
- Programs. For example, several programs that together satisfy one problem statement or make up one application.
- Networks. For example, several programs, each running

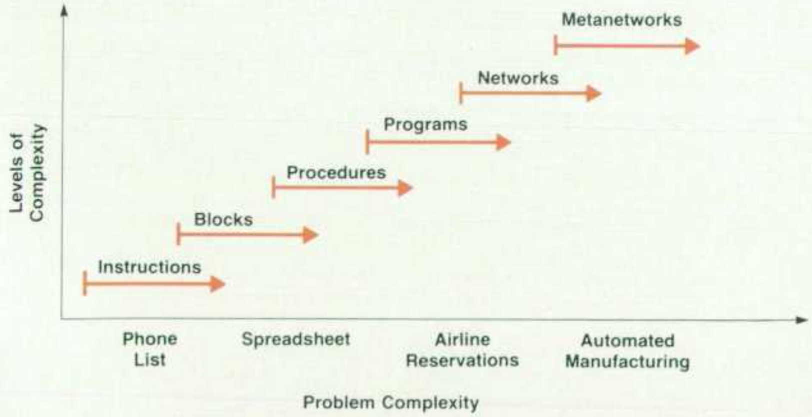


Fig. 2. *An intuitive comparison of program complexity, as scaled by modularity, with problem complexity.*

on a different computer, that together satisfy one problem statement.

In general, at each higher level on this scale new capabilities emerge. These permit richer relations and more powerful organizing principles to be designed into the software's architecture. Thus, a program can often be significantly simplified by modularizing at a higher level as shown in Fig. 3. The black curves in Fig. 3 intuitively describe how program complexity would grow as the number of modules grows. The curve in color shows how program complexity can be contained by using successively higher levels of modularization together with their emergent capabilities. It is exactly this effect that enables programmers to create ever larger programs that solve more and more complex problems.

Communications

A necessary companion of modularity is data exchange. To do useful work, modules must be able to share or exchange information. For each method of organizing modules into modules there exists one or more communication mechanisms among modules structured with the method. With the UNIX operating system and the C programming language, new communications capabilities emerge at every level of Fig. 2's scale.

Table I shows some major communication mechanisms associated with our module scale's modules. For example, a simple program shares data among its procedures, but not with other programs. Procedures use global variables, parameters, local variables and pointers, etc. to pass data. More complex problems can require coordinating the work of many computers at many sites. Networks, together with their communications protocols, connect programs running in separate computers. Between these examples are problems whose complexity exceeds what is appropriate for a single program, but that can be well-modeled by a few programs running on a single computer. Interprocess communication mechanisms allow data to be exchanged or shared and messages to be sent and received by such programs. IPC extends our ability to organize and structure programs.

IPC enables isolated simple programs to be integrated into multiprogram organizations. For example, a program that must respond to multiple asynchronous inputs will almost certainly be much simpler if modeled as several concurrent communicating programs. Each asynchronous input could have its own input handler program that transfers the input's data to a main program using IPC. The simplification derives from the ability to use UNIX resources (e.g., scheduler, interrupt handler, buffers, buffer manager, etc.) rather than recreate them in the application.

Table I

Module	Communication Mechanism
metanetworks	?
networks	network protocols, custom gateways
programs (processes)	interprocess communication, calling arguments, signals, files
procedures	global and external variables, parameters, common
blocks	arrays, structures,variables, array items, pointers, structure members
instructions	variables, array items, pointers, structure members

A final important point to note about communications mechanisms in general and IPC in particular: the mechanism is just that, a capability that makes communications possible. For useful communication to occur, the sender and receiver must also agree on rules and conventions for interpreting the data to be exchanged. These rules manifest themselves, in part, as a need for type agreement for within-program communication, format agreement for IPC, and protocol agreement for networks.

Concurrency

The UNIX system is a multitasking operating system. This means that the UNIX system can simultaneously execute many programs. To do so, it must keep information about a running program as well as the program's internal data. A process is what a program is called when it is actually running. A process is composed of a program's code and internal data plus the kernel-allocated resources needed to support the program at run time. For example, the kernel must remember where in main memory the pro-

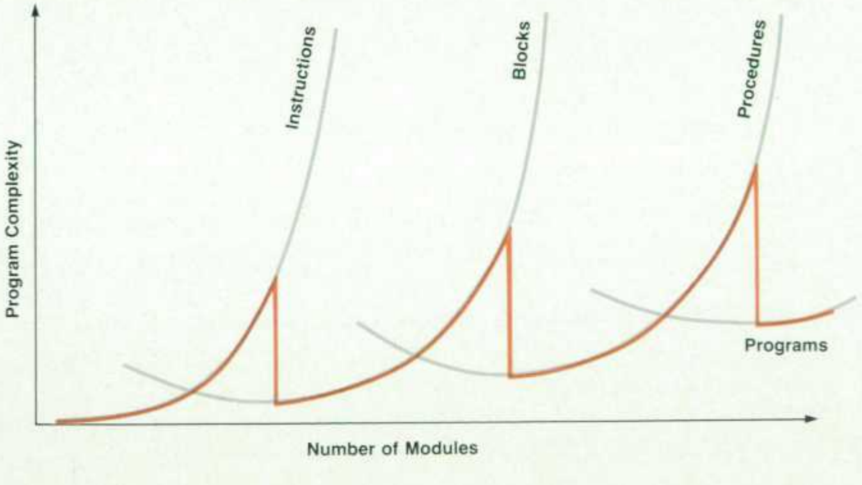


Fig. 3. *An intuitive comparison of program complexity with modularity. The color curve shows how program complexity can be reduced by modularizing at successively higher levels.*

gram is located, whether any buffers are being used, the current priority, permissions, etc.

Two or more processes running at the same time can be working in consort to solve a single problem. Usually, such multiprocess applications need to exchange data and information. The UNIX operating system provides IPC facilities for exactly this kind of information exchange. The UNIX kernel keeps track of any IPC resources used by such processes.

Every new process is created by the UNIX system in the same way: an existing process spawns an almost identical child process. Unless the original process makes special arrangements, the child process will have no knowledge of the parent or any other sibling processes that may exist. Some IPC mechanisms, notably pipes, require that the parent process set up the communications channel. This restriction causes some software systems to become excessively complex. Newer UNIX IPC mechanisms do not have this restriction. We will use the term process, rather than program, when we wish to emphasize the run-time nature of a program.

Synchronization

Concurrency introduces multiple execution threads that must be synchronized. For example, deadlock is a control problem that can occur when processes simultaneously reserve shared resources. Two processes can be waiting for an event—the other process to unlock the shared resource—that cannot occur because both processes are blocked waiting for the resource.

Concurrency aggravates problems with synchronizing data production and consumption. For example, suppose process p1 is sending messages (i.e., data) to process p2. They must agree to some convention that ensures that p2 starts reading only after p1 has sent a new message and that p1 begins writing only after p2 has read the old message. If p2 gets out of sync with p1, three situations can occur. One, p2 can read the same message twice; this is sometimes referred to as stale data. Two, p2 can miss reading a message; this is sometimes referred to as missed data. Three, p2 can read a partial or mixed message (i.e., p1 is preempted while writing a message, then p2 reads a message containing both new and old data); this is sometimes referred to as trashed data.

The potential for stale, missed, and trashed data exists even in simple programs. However, a single program's single execution thread implicitly imposes a strict sequentiality on data accesses. This usually guarantees correct data access synchronization. Data exchanges among concurrent processes must be carefully designed to avoid synchronization problems and achieve the benefits of multitasking.

It is interesting to note that here we have come full circle, as shown in Fig. 1. Synchronization problems are a new form of program complexity that must now be dealt with. It seems reasonable to ask why facilities that introduce complexity, such as IPC, are useful. Fundamentally, IPC exists because it can be used to trade a little more complexity in handling data for much less complexity in structuring programs. This trade-off arises, in large part, because asynchronous events can be much more easily incorporated into a multiprogram design than into a single monolithic program. That is, synchronizing data exchange is easier and less problematic than synchronizing control events.

UNIX IPC and Software Architecture

One aspect of developing concurrent programs is identifying communication mechanisms that match the structural relationships of the software architecture.

The UNIX operating system provides several IPC facilities. Unfortunately, no one UNIX IPC facility is best for every possible application. The best IPC method for a given application depends on the structure of the communicating programs, the amount and kind of data that must be passed, the performance that the application demands, and the capabilities of the underlying hardware. Since each mechanism can be used, or adapted, for many purposes, evaluating each IPC facility against specific requirements is usually necessary.

IPC Use Taxonomy and Ranking

A simple taxonomy for comparing the UNIX IPC facilities according to the uses for which they seem best suited is shown in Fig. 4.* In this scheme, IPC uses are classified

*Some UNIX versions, for example, Berkeley 4.2 BSD, contain IPC facilities that UNIX System V does not. These are not discussed in this paper, but are described in references 1 and 2.

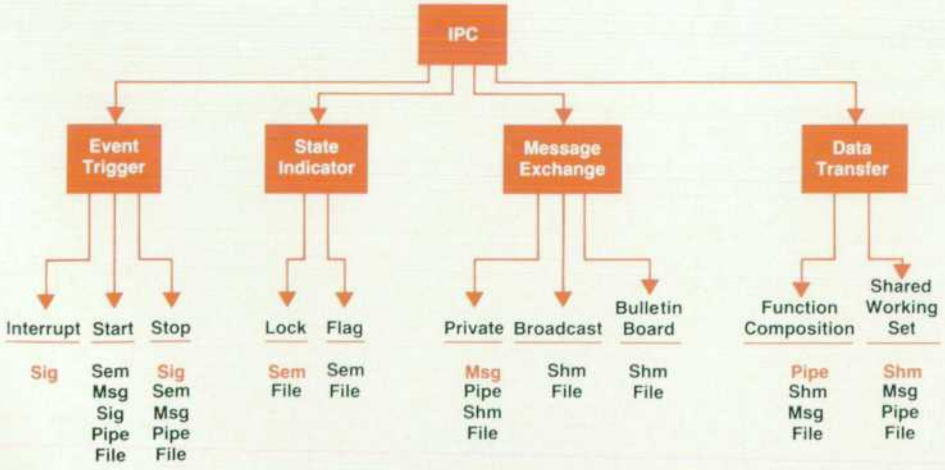


Fig. 4. *IPC use taxonomy and facility ranking. Msg stands for message queues, Pipe for pipes, Sem for semaphores, Shm for shared memory, and Sig for signals. The use for which a facility seems to have been designed originally is shown in color.*

into four broad classes, each containing two or three subcategories. The UNIX IPC mechanisms are then ranked under these subcategories from top to bottom, from best suited to worst suited, respectively. These rankings and taxonomy should be interpreted as suggestions. No claim is made for their universal applicability. Indeed, they include at least one seeming paradox. Namely, even though shared memory is theoretically the fastest possible IPC facility, it is suggested that it not be used for those purposes demanding quick delivery. Hopefully, the material presented below will make the reasons clear.

Briefly, the uses the taxonomy's classes are trying to capture are as follows. The class of event triggers initiates some action, usually starting or stopping processing. The interrupt event trigger (i.e., the UNIX signal facility) forces the communication receiver to branch to a specified location or terminate. State indicators announce a condition, usually the accessibility or state of some resource. Message exchanges convey information, usually between two processes but occasionally from one process to many processes. Exchanges can be sent in rapid succession to specific receivers or posted and picked up by arbitrary receivers on an as-needed basis. Data transfers move results for successive operations; usually each function operates on the data just once but occasionally functions may alternate working with the data.

There are many different dimensions to the taxonomy's use categories. They can be characterized by the nature of their requirements. Fig. 5 compares each use's important requirements with the properties of the UNIX IPC facilities. The goodness of the match, together with each facility's throughput (discussed below), form the basis for the rankings in Fig. 4. The properties of Fig. 5 are:

- Quick delivery—the elapsed time between posting a communication and its availability is small.
- Atomic operation—partially completed results of an interrupted or preempted operation cannot be accessed by another process.
- Multiple readability—a communication can be read repeatedly until it is purposely changed.
- Universal readability—any process with the proper permissions can access a communication.
- Fair capacity—a given communication can pass a reasonable amount of data, say a few thousand bytes.
- High capacity—a given communication can pass substantial amounts of data, say many tens of thousands of bytes.
- No content constraints—a communication can have arbitrary content and/or any user-defined format.

The following properties listed in Fig. 5 are particularly useful in comparing IPC facilities (other, less important properties, are left to each facility's discussion later):

- Simplicity of use—one does not need to refer to manuals or take extreme care in design to use the facility successfully.
- Generality of connection—a communication's receiver's actions on the communication do not directly affect its sender.
- Multiple setup calls—more than one system call is required to create a communication link.
- Minimum CPU resources—a given communication consumes virtually zero central processing unit (CPU) cycles

Property

	Use and IPC Facility Properties							IPC Facility Properties Only						
Use:	Quick Delivery	Atomic Operation	Multiple Readability	Universal Readability	Fair Capacity	High Capacity	No Content Constraints	Simplicity of Use	Generality of Connection	Multiple Setup Calls	Minimum CPU Resources	Needs Other IPC Facility	Directly Affects Control	
Interrupt Start, Stop	X	X		X										
Lock		X	X	X										
Flag			X	X										
Private	X	X		X	X									
Broadcast	X	X	X	X	X									
Bulletin Board		X	X	X	X									
Function Composition				X		X	X							
Shared Working Set			X			X	X							IPC Facility:
				o					o				o	Kill/Signal
	o	o	o	o						1				Semaphore
	o	o		o	o		o	o	o					Message Queue
	2		o	o	o	o	o			o	3	o		Shared Memory
	4				o	o	o	o	o					Pipe
			o	o	o	o	o	o	o					File

Notes:
X Properties this use requires.
o Properties this IPC facility exhibits.
1. Semaphores used as resource locks usually require value initialization.
2. Theoretically the quickest; in practice, two semaphore operations are needed.
3. Minimum achieved if and only if users always keep attached.
4. Achieved if and only if data is found in buffer.

Fig. 5. *Use requirements and IPC facility properties.*

Benchmarking UNIX IPC Facilities

The usual purpose of benchmarks is to quantify different computers' performance in a way that permits valid comparisons among them. Typically, the goal is to discover price/performance ratios that permit near-optimal purchase decisions. Our benchmark serves a similar purpose. Although it attempts to quantify UNIX IPC performance in a way that permits valid comparisons, the goal is to create performance/capacity curves that provide insight into the relative effectiveness of the UNIX system's IPC facilities. We hope that this will permit better system/software design decisions.

The good news is that since the measurements were made on the same computer, most standard benchmarking perils are avoided. The bad news is that the IPC facilities have features that are incomparable. This benchmark is by no means a complete characterization of an IPC facility's capabilities. In particular, this benchmark uses a message-passing paradigm. Thus, it accurately measures IPC mechanisms with respect to message exchange. Other uses, however, require some care in interpreting the results.

We believe that event triggers and state indicators can be fairly treated as passing a single data element. So the performance of semaphores, signals, and messages can be fairly compared for the special case of a message size equal to one. Data transfers, however, are not so easily handled.

There are two important considerations not addressed in this benchmark. One, shared memory can pass data without copying it. For organizations in which processes alternately work directly in shared memory (e.g., a data base and its query processes or a windowing suite), the performance of shared memory could greatly surpass any other IPC alternative. Two, pipes use the UNIX operating system's file buffers. On a system where buffers are a scarce resource (e.g., a heavily loaded I/O-intensive system), the performance of pipes would probably decrease. Consequently, this benchmark underrates shared memory's performance and overrates that of pipes.

Benchmark System

All tests were run on an HP 9000 Series 320 Workstation with 4M bytes of main memory. (This system is based on a 16.6-MHz MC68020 CPU with a 16K-byte cache, 12.5-MHz MC68881 floating-point coprocessor, and memory management unit.) The operating system was HP-UX Multiuser Revision 5.1. (HP-UX is composed primarily of AT&T Bell Laboratories' UNIX System V.2, but with Hewlett-Packard's own extensions and features from the 4.1 and 4.2 BSD versions of the UNIX operating system by the University of California at Berkeley.) All benchmarks were run in single-user mode. (Processes 0 (swapper), 1 (/etc/init), and 2 (pagedaemon) were the only processes running besides the two benchmark processes.)

Benchmark Programs

All benchmark programs had the same general structure. Pseudocode for the shared memory benchmark would look as follows:

```
obtain number of messages to send and the length of a message
fork child process to read messages
        get and attach shared memory segment for child
        get memory-filled and memory-empty semaphores
        loop forever
                acquire memory-filled semaphore, block if no data
                copy data from shared memory into buffer
                release memory-empty semaphore
fill parent's buffer with message
get and attach shared memory segment for parent
get memory-filled and memory-empty semaphores
start timer
loop N times, where N = number of messages to send
        acquire memory-empty semaphore, block if data unread
        copy data from buffer into shared memory
        release memory-filled semaphore
stop timer
return memory and semaphores to system
print results
```

The semaphore benchmark differed from the above in that the processes only acquired and released semaphores. No data was passed. In the message queue benchmark, the parent and child alternated sending a message and reading a reply with blocking.

The pipe benchmark was similar to the message queue benchmark except that two pipes were used. The signal benchmark was similar to the semaphore benchmark except that signals and handlers were used to escape pauses alternately.

Copies of these benchmark programs are available from the author. The results are provided in the accompanying article (Fig. 6).

Caveat

No useful work was done when a benchmark program was running. One hundred percent of CPU time was spent in handling messages. Thus, this benchmark gives absolute upper limits on this system's IPC capabilities. The practical limits are somewhat lower.

to complete.

- Needs other IPC facility—the normal use of the facility requires other IPC facilities.
- Directly affects control—the receiving processes' execution path is interrupted by the communication.

Performance

Capacity. The use of a facility dictates its communications' range of sizes, that is, the number of bytes that a communication needs. The classes in Fig. 4 require greater capacity as one moves from left to right. For example, data transfers usually require an IPC mechanism with significantly greater capacity than do event triggers.

The capacity of a particular UNIX facility is usually set when the system is configured. It will vary from installation to installation. However, Table II presents some typical data for the capacities of UNIX IPC facilities. This data is for the same system described in the box above.

Throughput. The UNIX operating system makes no guarantee on the length of time it takes to deliver a communication—forever is not impossible, although somewhat unlikely. Indeed, one has to work very hard to delay a message

Table II

IPC Facility	Maximum Capacity
file	size of file system
pipe	48K bytes (blocks at 4K bytes)
FIFO	same as pipe
message queue	8K bytes
shared memory	4M bytes
signal	1 signal from set of 23
semaphore	1 value from ±32K

more than a few hundred milliseconds. This is true for all IPC mechanisms. On the average, though, some are capable of delivering more messages per unit time than are others. Of course, a communication's size interacts with a facility's capacity to affect its performance. Fig. 6 presents some comparative data on UNIX IPC performance. A portion of the data shown in Fig. 6 appears to defy the conventional wisdom. In particular, shared memory is not found to be the best performer.

The surprising results of Fig. 6 appear to be an artifact of using semaphores and shared memory for purposes for which they are not well-suited. Our taxonomy, Fig. 4, suggests that semaphores are best used as state indicators and shared memory is best used for data transfer. However, the bencnmark program combines these two facilities into a message exchange medium. The box on page 31 explains how the data for Fig. 6 was obtained.

Blocking. It is possible to block on most IPC service calls. That is, the kernel suspends the process—halts its consumption of CPU cycles—until the desired external event occurs. Thus polling or busy waiting, which consume CPU cycles, are unnecessary. If more than one process is blocked on a particular IPC facility, the UNIX system makes no guarantee as to which process will obtain the facility when it unblocks. For example, the UNIX system guarantees that message queue communications will be delivered in a first-in-first-out (FIFO) order, but does not guarantee that processes will get them on a first-come-first-served basis. It is also possible to continue without delaying on most IPC service calls.

UNIX IPC Facilities

The traditional UNIX IPC facilities are files, pipes, FIFOs (named pipes), and signals. Pipes, FIFOs, and, of course, files all use the file system to pass data. They pass information as byte streams, the natural interpretation of all information in the UNIX file system.

Files

Files are not truly IPC facilities, but they can be and often are used for IPC. Two or more programs that use a file for IPC must agree on conventions for data format and synchronization. Typically, the presence or absence of a second lock file is used to signal when the data file is being accessed.

The UNIX system buffers file input/output and uses de-

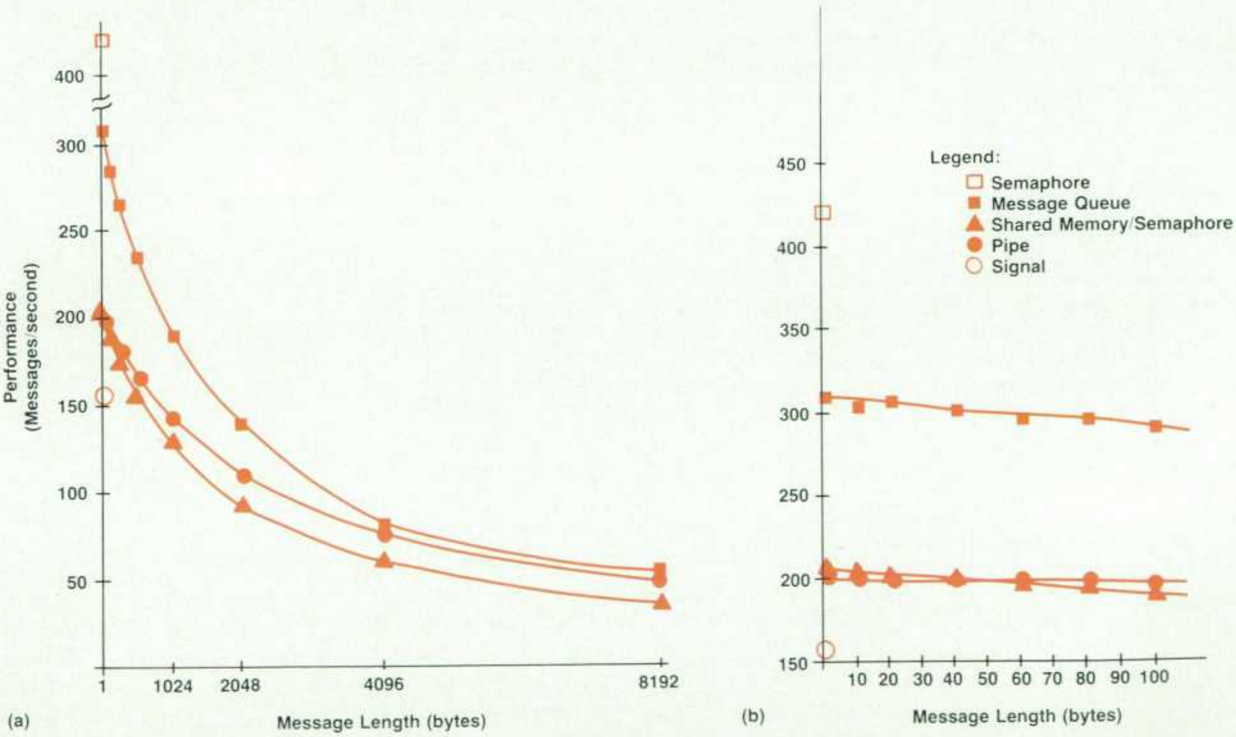


Fig. 6. *The relative performance of UNIX System V IPC facilities. (a) The relative performance for long messages. (b) The relative performance for short messages.*

layed writes during output. So, for short communications sent during periods of light loading, file IPC can be fairly efficient (since the data will often be found by the receiver in the buffer). However, for long messages and/or heavily loaded systems, file IPC will be the least efficient mechanism possible. The advantages of file IPC are unlimited capacity and multiple delivery. A single message can contain as many bytes as the largest legal file. A single message can be read by more than one process.

Pipes

The classic UNIX IPC mechanism is the pipe. Fig. 7 shows how pipes work. Fig. 7a depicts a pipe in complete generality while Fig. 7b shows the preferred arrangement for two communicating processes. Pipes enforce a rigid first-in-first-out order to messages. One or more processes can write into a pipe and one or more can read from it. However, each message can be read only once. These processes must have a common ancestor (who sets up the pipe). Normally, writes to a full pipe block the writing process until a reader removes some data while reads from an empty pipe block the reading process until there is data to read.

The performance of pipe IPC will be the same as file IPC, up to a point. Pipes are restricted to use only a small part of a file's potential capacity; this prevents some expensive disc accesses. This is more efficient from a systems viewpoint, although it might force a longer delay in delivering a message.

Synchronization of communications is automatic and reliable if a pipe has exactly one writer and one reader. Since writes and reads to or from a pipe are not atomic (i.e., guaranteed to finish before preemption), reliability remains an issue if there are more than two writers or readers.

Pipes are a very general mechanism. They can be used for many purposes. However, they seem best suited to perform function composition, successively passing unstructured data from one process to another.

FIFOs

Named pipes, or FIFOs, are identical to pipes in operation. FIFOs, unlike pipes, are not constrained to have the channel set up by a common ancestor. A process need only know a FIFO's name and have permission to access its contents.

Signals

Another classic UNIX IPC mechanism is the signal. Signals are usually used to trigger events. Signals are usually constrained to use a single value from a small set (about 20) of predefined values. This seriously limits their ability to pass information. Signals are not queued and there is no indication of who the sender is. Signals, despite their simplicity, are not very efficient IPC mechanisms.

Historically, signals were created as a mechanism to terminate processes. There are two problems with using signals for IPC: it is possible for signals to be lost and it is possible to terminate a receiving process prematurely. Both problems occur if signals of the same type are received at too high a frequency.

Unlike other IPC facilities, signals affect control directly rather than simply exchanging data. They should be used carefully, if at all.

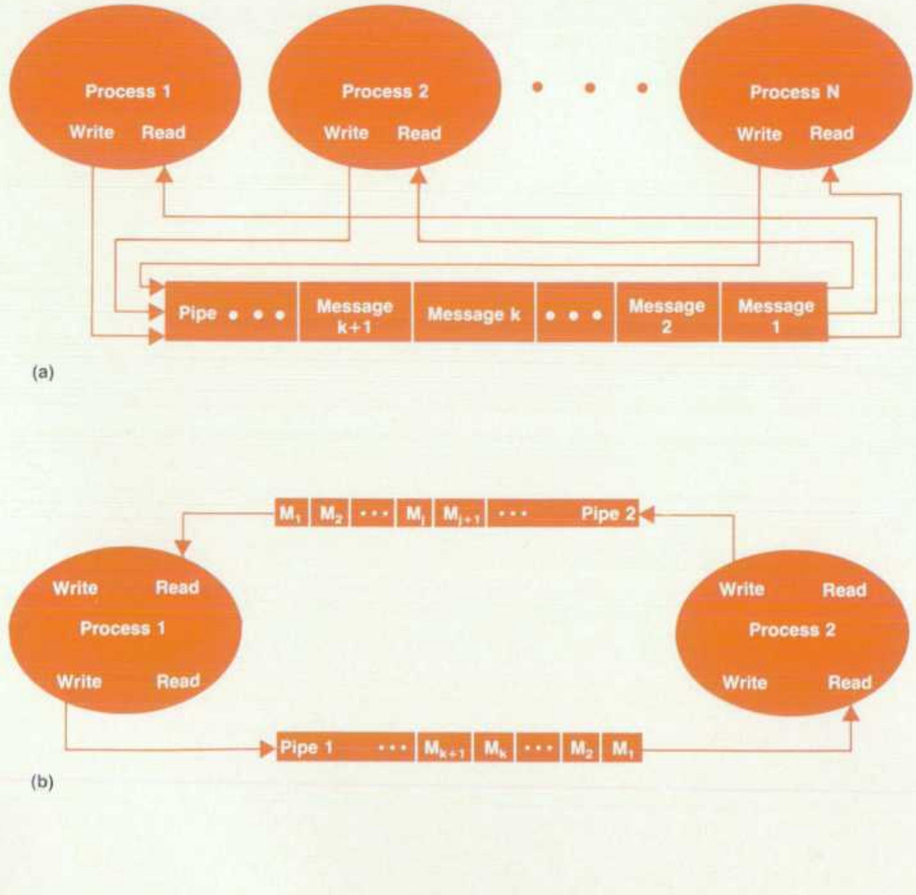


Fig. 7. *The structure of pipes. (a) General structure. (b) Preferred pipe structure for bidirectional communication between two processes.*

System V IPC Mechanisms

Pipes and signals permitted software solutions not practical with single monolithic programs. The new IPC mechanisms in UNIX System V permit, in turn, more sophisticated solutions than was reasonably possible with pipes and signals. UNIX System V adds three very powerful IPC facilities: message queues, shared memory, and semaphores. These facilities do not use the file system or require common ancestry. Thus, program design constraints are much reduced. All are accessed in the same way: a process that knows the right key and has the right permissions requests the kernel to connect it.

Semaphores. Semaphores are used to control access to shared resources, to synchronize events, and/or to announce the state of some object. They are generally used as indicators, rather than to pass information. Fig. 8 shows how semaphores might be used to protect a resource. Conceptually, they use a single machine word. In fact, however, they are somewhat more complicated and UNIX System V does considerable processing to effect a semaphore transaction.

Semaphores must often be used in pairs: once to protect or lock a resource and once to release or unlock it. For example, consider the use of semaphores with shared memory to create a message-passing medium as described earlier and in the box on page 31.

A single semaphore system call is the UNIX system's fastest IPC facility.

Message Queues. Message queues are most frequently used to exchange messages. They behave similarly to pipes in that messages are passed on a first-in-first-out basis. However, associated with each message is a type number and an argument. The type number enables more flexible program designs. Successive messages with the same type form queues, but a receiver can choose which type(s) to read. The argument is an array of characters, but not quite a C string (there is no guarantee of a null terminator).

A message can be read only once. Each access of a message queue is atomic. This eliminates one source of unreliability using pipes. It is possible to read a queue without blocking if it happens to be empty. Any number of processes can access a queue, and they need not be related. A process can access a queue if it has the correct permission and has the queue's key. Fig. 9 shows how message queues work.

There are three primary strategies for using type numbers: as mailbox IDs as shown in Fig. 9a, as priorities as shown in Fig. 9c, or as op codes. If all messages sent via a particular message queue have the same type number, then the queue is a strict first-in-first-out byte stream. If messages have different type numbers associated with them, then each type acts as a first-in-first-out byte stream. Consequently, messages can be read in an order different from their write order.

Message queues are a very general mechanism, and can be used for many purposes. They seem best suited to conveying small, private communications from one process to another. Message queues are probably the fastest practical IPC mechanism for this purpose.

Shared Memory. Shared memory is most frequently used to transfer data from one process to another. Shared memory uses coordinated writing and reading to and from the same physical memory locations to share working data among separate processes. That is, two or more distinct programs can arrange to have the same set of buffers and/or variables, much like global common variables or buffers for a single program. To pass data reliably using shared memory requires synchronizing data writes and reads. This is often accomplished with semaphores. Fig. 10 shows how shared memory works.

Shared memory can pass arbitrarily complex information. Capacity is limited only by the amount of main memory available. It and semaphores are special in that more than one process can read the same communication. Shared memory is theoretically the fastest possible IPC mechanism. In practice, shared memory is sensitive to the use to which it is put.

Although data is sent at the instant it is written in memory, receiving the data is somewhat more problematic. If many processes are competing for the same memory, then

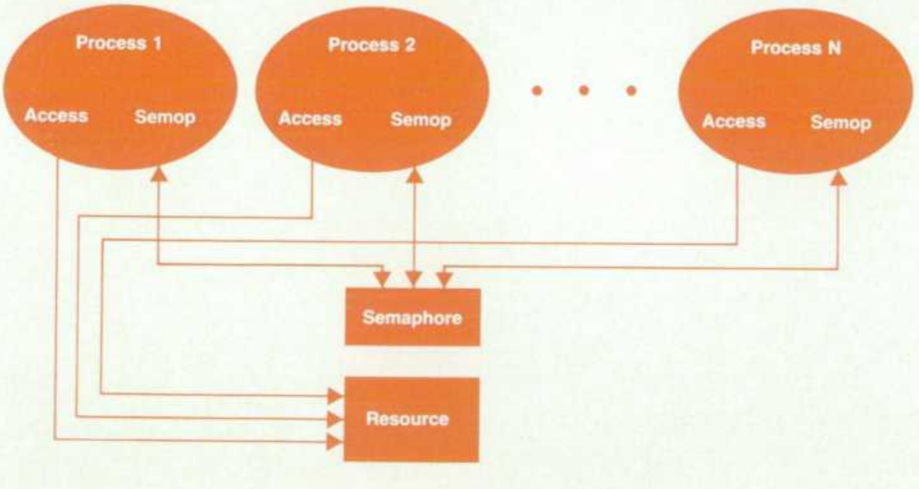


Fig. 8. *The use of a semaphore to protect a system resource. Semop is a semaphore operation. Decrementing a semaphore acquires it if it was free, and thereby locks the resource. Incrementing a semaphore releases it if it was acquired, and thereby unlocks the resource. While one process uses (locks) the resource, the semaphore blocks the other processes. When the process using the resource completes, it releases (unlocks) the resource for use by other processes.*

they may be forced to copy the shared memory contents to prevent blocking each other. Since shared memory accesses are not atomic, processes must signal both when new data has been written and when old data has been read. In general, this implies either two semaphores or two lock states for the same semaphore and bracketing semaphore operations in both the sending and receiving processes.

Shared memory is a general mechanism. It can be used for many purposes. It seems best suited to sharing large working sets of data between alternately executing processes. However, shared memory tends to reintroduce exactly the kinds of program coupling that historically have created problems. Shared memory should be used with some care.

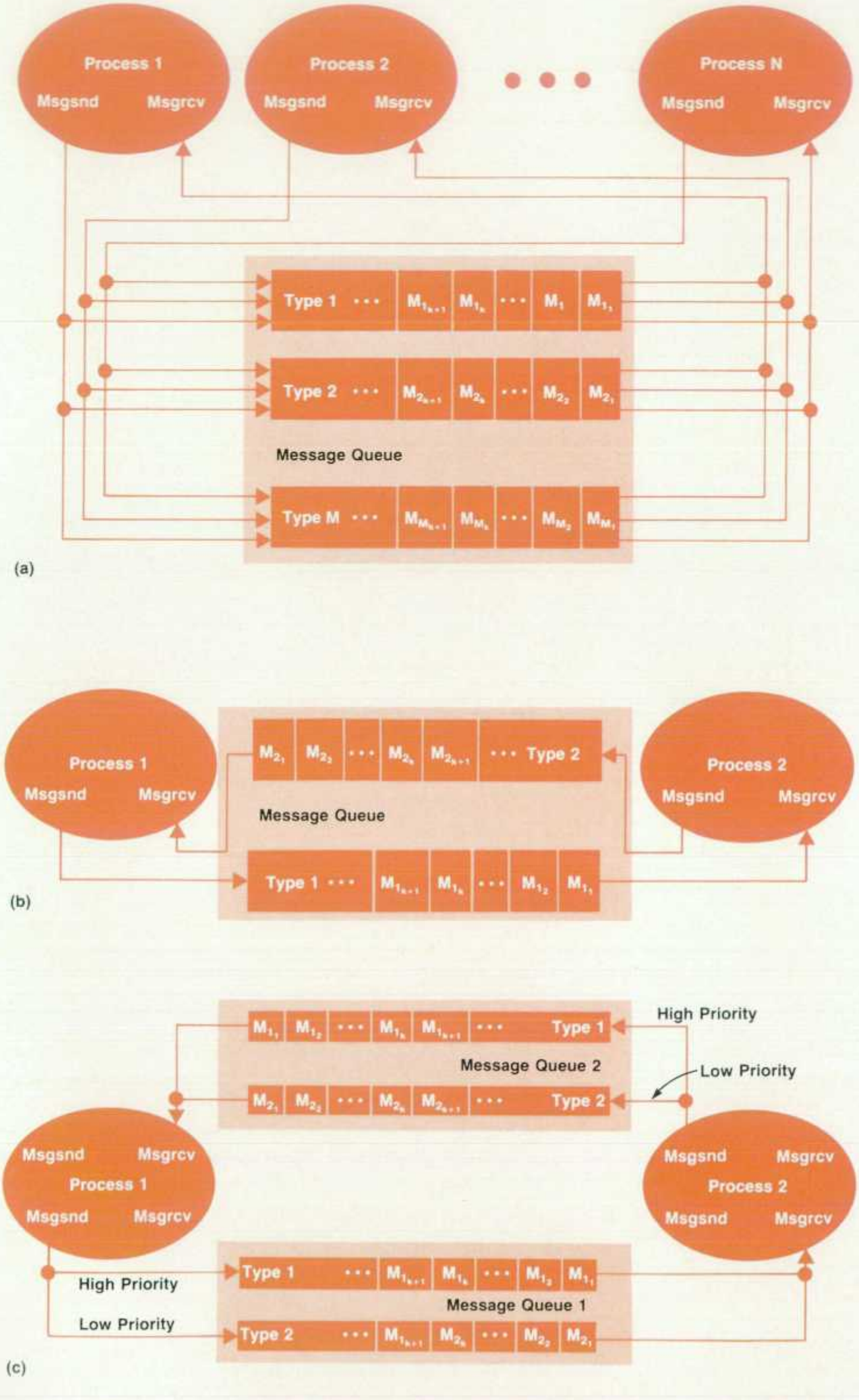

Process 1
Msgsnd
Msgrcv
Process 2
Msgsnd
Msgrcv
• • •
Process N
Msgsnd
Msgrcv
Type 1 ••• $M_{1_{k+1}}$ M_{1_k} ••• M_{1_2} M_{1_1}
Type 2 ••• $M_{2_{k+1}}$ M_{2_k} ••• M_{2_2} M_{2_1}
Message Queue
Type M ••• $M_{M_{k+1}}$ M_{M_k} ••• M_{M_2} M_{M_1}
(a)
Process 1
Msgsnd
Msgrcv
M_{2_1} M_{2_2} ••• M_{2_k} $M_{2_{k+1}}$ ••• Type 2
Message Queue
Type 1 ••• $M_{1_{k+1}}$ M_{1_k} ••• M_{1_2} M_{1_1}
Process 2
Msgsnd
Msgrcv
(b)
M_{1_1} M_{1_2} ••• M_{1_k} $M_{1_{k+1}}$ ••• Type 1
High Priority
Message Queue 2
Low Priority
M_{2_1} M_{2_2} ••• M_{2_k} $M_{2_{k+1}}$ ••• Type 2
Msgsnd
Msgrcv
Process 1
Msgsnd
Msgrcv
Msgsnd
Msgrcv
Process 2
Msgsnd
Msgrcv
High Priority
Type 1 ••• $M_{1_{k+1}}$ M_{1_k} ••• M_{1_2} M_{1_1}
Message Queue 1
Low Priority
Type 2 ••• $M_{1_{k+1}}$ M_{2_k} ••• M_{2_2} M_{2_1}
(c)

Fig. 9. *The structure of message queues. (a) General structure. (b) Typical structure for bidirectional communication between two processes. (c) Typical structure for bidirectional communication between two processes with two priorities, high and low. Msgsnd is a message send (transmit) operation which places a message into the message queue. Msgrcv is a message receive operation which retrieves the specified message, usually the next, from the message queue.*

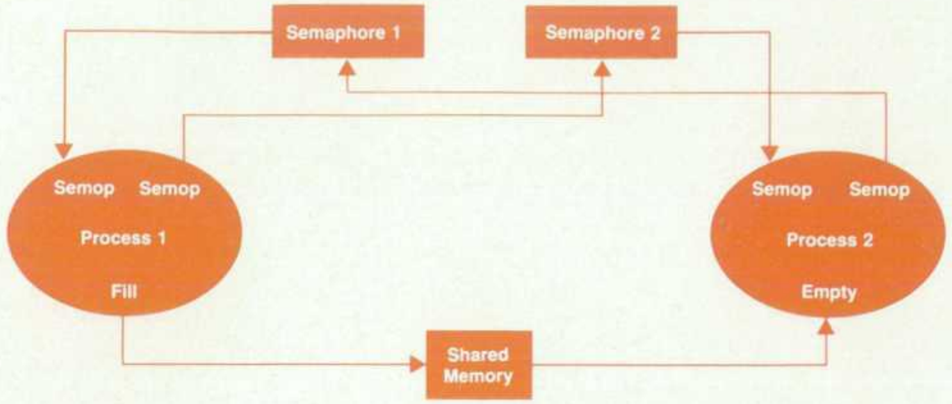


Fig. 10. *Typical shared memory structure for bidirectional communication between two processes. Semaphores are used to coordinate message transfer. Semaphore 1 blocks process 1 while process 2 empties shared memory. When shared memory is empty, process 2 releases semaphore 1, unblocking process 1 and unlocking shared memory. Process 1 then sets semaphore 2 to lock shared memory and block process 2 while process 1 fills shared memory. When shared memory is full, process 1 releases semaphore 2.*

Acknowledgments

The author would like to thank Kyle Polychronis and Irwin Sobel for their valuable comments and the generous contibution of their time in reviewing this paper.

References

1. M. Rochkind, *Advanced UNIX Programming*, Prentice-Hall, 1985.
2. R. Thomas, et al, *Advanced Programmer's Guide to UNIX System V*, McGraw-Hill, 1986.
3. M.J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.
4. *HP-UX Reference, Vol. 2: Sections 1M and 2*, Hewlett-Packard Company, 1985.

Hewlett-Packard Company, 3200 Hillview
Avenue, Palo Alto, California 94304

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

HEWLETT·PACKARD JOURNAL

June 1987 Volume 38 • Number 6

Technical Information from the Laboratories of
Hewlett-Packard Company
Hewlett-Packard Company, 3200 Hillview Avenue
Palo Alto, California 94304 U.S.A.
Hewlett-Packard Central Mailing Department
P.O. Box 529, Startbaan 16
1180 AM 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 subscribe, change your address, or delete your name from our mailing list, send your request to Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304 U.S.A. Include your old address label, if any. Allow 60 days.

5953-8561